1-1-1980

Integrated testing methodology for system development projects.

Linda Khatri

INTEGRATED TESTING METHODOLOGY FOR SYSTEM DEVELOPMENT PROJECTS

by

Linda Khatri

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1980

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

August 27, 1980
(Date)

Professor in Charge

Chairman of Department

ACKNOWLEDGMENTS

I wish to express sincere gratitude to Professor Larry E. Long for the inspiration, the patience, and the interest shown through several revisions of this thesis. His help and guidance were truly indispensable.

I would also like to thank all those persons who responded to my request for information regarding software testing packages.

Finally, I am indebted to several friends and colleagues for reading this and for making many important corrections and helpful suggestions.

CONTENTS

List of Figures

ABSTRACT

A methodology is proposed for testing a computer-based information system throughout its development process. Normally, testing is performed after a system has been coded and debugged, if at all, and ad hoc procedures are relied upon to perform the process. Little attention has been paid towards quality assurance, and as a result, more effort is devoted to system maintenance than is to system development. Some reasons identified as contributing to this are poor attitudes towards testing, lack of planning, and little knowledge or understanding of test theory. This proposed methodology makes testing integral to system development, prescribing a series of checks which parallel a typical system development process. It provides for creation of a test team similar to a chief programmer team, and describes the training and career path of a tester. Tools and techniques available to the tester, such as static code analyzers and test data generation methods are explored, and recommendations made for their selection. Implementation of this methodology will solve many of the aforementioned problems by creating a more controlled environment in which responsibilities and procedures are clearly defined. One benefit might be a substantial reduction in the percentage of a system's life cycle to be devoted to maintenance.

Chapter I. Introduction

A. Purpose of this study

Of the entire software development process, perhaps the least developed area is that of testing. Paradoxically, it is also one of the most critical steps. Post-installation maintenance of systems is estimated to account for up to 70% of the total software effort [Fairley, 1978]. This time could be substantially reduced with a well thought out and well designed test plan, which, when followed during software development, would assure more reliable systems.

There is clearly a need for improved testing methodology and procedures in the software development stage. Rather than relying on ad hoc methods, the manager of a project should have a well-defined view of the necessary steps in the process and should impose standards for determining software reliability. Test procedures are as important to the process as system design procedures, and should be treated accordingly.

The purpose of this study is to propose a standardized methodology which can be implemented in any software development environment. Chapter two outlines this methodology, describing where and how it fits into the

system development process. In chapter three specifics of the proposed methodology are discussed from a managerial point of view, explaining why certain methods and techniques have been selected over others and detailing means of implementing the proposed plan. Chapter four describes some tools and techniques available to the tester and provides recommendations for tool selection. Finally, in chapter five, the methodology is summarized in terms of its benefits to the system development process.

B. Literature survey

The literature of software testing is sparse compared to that of other computer related activities. A couple of hundred articles and books have appeared since 1973. The bulk of these deal with test theory and individual program testing rather than test management procedures. A review of the literature indicates the fact that much of the research has not been brought out of the labs and into the field, where it can be applied. Only one book has appeared to date which is totally related to program testing [Myers, 1979]. However, its treatment of subjects such as system testing is limited, and project test management is hardly discussed at all. The main objective of the book is to provide a

programmer or program tester with general guidelines to follow in testing a single program or series of programs.

1. Test management procedures

Almost all of the literature relating to test management is concerned with general subjects. Most good software development or DP management books will contain a chapter on testing, but none prescribe a specific methodology (other than the top-down or bottom-up approach), any means for test reporting, criteria for determination of test completion, etc. Testing is normally viewed as a single step in the software development process.

Much of the general groundwork has been done by Miller [1978], who discusses some organizational schemes, justifications for testing, the psychology of testing (frequently maintained attitudes), and budget allocations. Some work has been done on other management issues, especially recommended formats for test planning and reporting [Mullin, 1977], [Buckley, 1973], [Bate, 1978], [Hartwick, 1977], [Krause, 1978]; however, there is no single prescribed methodology for managing the software testing process from beginning to end in detail. Myers and Fagan propose means for organizing a project team to perform code inspections and walkthroughs [Myers, 1979], [Fagan,

1976]. With regard to the amount of time and effort to be devoted to testing, there is some mention of recommended percentages of the total software effort [Alberts, 1976], [Boehm, 1975], but no one has proposed a specific means of incorporating these factors into the software development process.

2. Test theory and methodology

More abundant than the literature related to test management is that involving program test theory and methodologies. Program testing is normally classified as static or dynamic, depending upon what is known about the internal structure of the program and the type of information desired. Static analysis is used to obtain global information regarding program structure, such as uninitialized variables, variable cross-references, etc., without regard for run-time behavior. The purpose of static analysis is to demonstrate the truth of a structural, syntactical, semantical, or interprocedural allegation. Dynamic testing, on the other hand, involves executing the program in a controlled and systematic way to demonstrate that required functions are present and that unwanted functions are absent. Most of the literature discusses methods and theory of dynamic analysis, particularly of test

case selection to completely exercise a program.

The key paper regarding test data selection is that by Goodenough and Gerhart, which builds a set of mathematical theorems which can be useful in determining whether the selection of data used to exercise a program is successful, reliable, and valid. If the test data is selected to exercise every test predicate and if the program executes correctly for the set of all test data, then the program is correct [Goodenough, 1975]. This has since been referred to as the "Fundamental Theorem of Testing" [Howden, 1978e].

Almost all of the literature concerned with program testing theory discusses path testing via decomposition of the program control flow into a directed graph, where each node represents a transfer of control (branch) and each edge a portion of code between branches. Test data is normally selected to somehow exercise each of the branches at least once. None of the other theories is as complete as that of Goodenough and Gerhart, nor do they contain any new and extraordinary contributions to the general theory.

With regard to test data selection, the notion of symbolic execution of a program using algebraic symbols rather than real values is rapidly catching hold in academic circles [King, 1976], [Darringer, 1978]. Proponents of

symbolic execution claim that a single set of symbols can replace an entire class of input data and that an algebraic expression is easier to evaluate than a numeric value.

Another new idea is that of program mutation [DeMillo, 1978], [Budd, 1978] where mutated versions of the same program are executed with the same test data, and the results compared.

System test theory is less developed than program test theory, and the methodology less well-defined. Some literature exists on module testing, module integration (top-down vs. bottom-up especially), and system testing, but all these areas are generally treated merely as a small segment of the system development process. Most texts say that testing should be done, but few prescribe how to do it.

3. Testing tools

The literature describes several individual tools which can be used to aid in testing, especially with dynamic program testing. Many of these tools are experimental or otherwise not commercially available. Though the tools perform a variety of functions when taken together, most of them are relatively simple when considered individually, performing only one or two functions.

C. State of the Art

The basic philosophy of program testing has already been relatively well established in both practice and theory. Graph-theory has been used extensively to model both program control flow structure and data flow. Some tools and techniques exist for program analysis, test data generation, program instrumentation, etc., though they are not highly developed. Most of them perform only a few functions, and there is no single source for tools which apply to all categories of analysis for one particular language or system. In addition, many of the tools are language dependent.

Although much of the theory has been developed in research laboratories and universities, the next critical step is to bring the theory into the field. This involves some coordination of effort among all those involved or affected, and implies:

1. Standardization of methodologies;

2. Incorporation of testing into the software development process;

3. Development of usable tools which are commercially available.

System testing methodologies and principles are not as well developed or well established as those for program testing. Although testing is normally done in some fashion, too much time is spent in post-installation maintenance of the system for one to believe that it is being done methodically or well. Admittedly, not all of that time is attributable to poorly designed or untested systems; user requirements may have changed as well, and it is difficult to delineate what percentage of maintenance is actually spent in redesign or recoding to conform to original user requirements. The methodology proposed here could potentially alleviate the maintenance problem by reducing the time spent in post-installation maintenance. The amount of effort to be expended in software repair should decrease with time, leaving the developer free to consider new user requirements.

Chapter II. Integrated System Testing Procedure

A. Overview

The methodology proposed here consists of eleven phases, each of which coincides with a major step in the system development process. Proceding from a very broad perspective with the establishment of an overall general test plan, it narrows down as the elements under scrutiny become more and more detailed, and then widens again as the individual elements are built up to create a whole system.

The early phases begin with planning the overall testing process and examining general system design documents. As the elements of design become more detailed, so does the test planning and the actual testing. Testing reaches its narrowest perspective at the point where individual units of code undergo minute scrutiny by individual testers. Then, as the system is constructed from its parts, piece by piece, the testing team broadens its perspective until they are examining a whole system, first within itself and then against its intended functions. The final step represents a summation of the entire process, so that the techniques and procedures can be refined and improved for future projects. Figure 1 graphically depicts

I. G E N E R A L T E S T P L A N

II. GENERAL SYSTEM DESIGN ANALYSIS

III. DETAILED TEST PLAN

IV. DETAILED SYSTEM DESIGN ANALYSIS

V. UNIT TEST PLAN

VI. INTEGRATION TEST PLAN

VII. UNIT TEST

VIII. INTEGRATION TEST

IX. PERFORMANCE TEST PLAN

X. P E R F O R M A N C E TEST

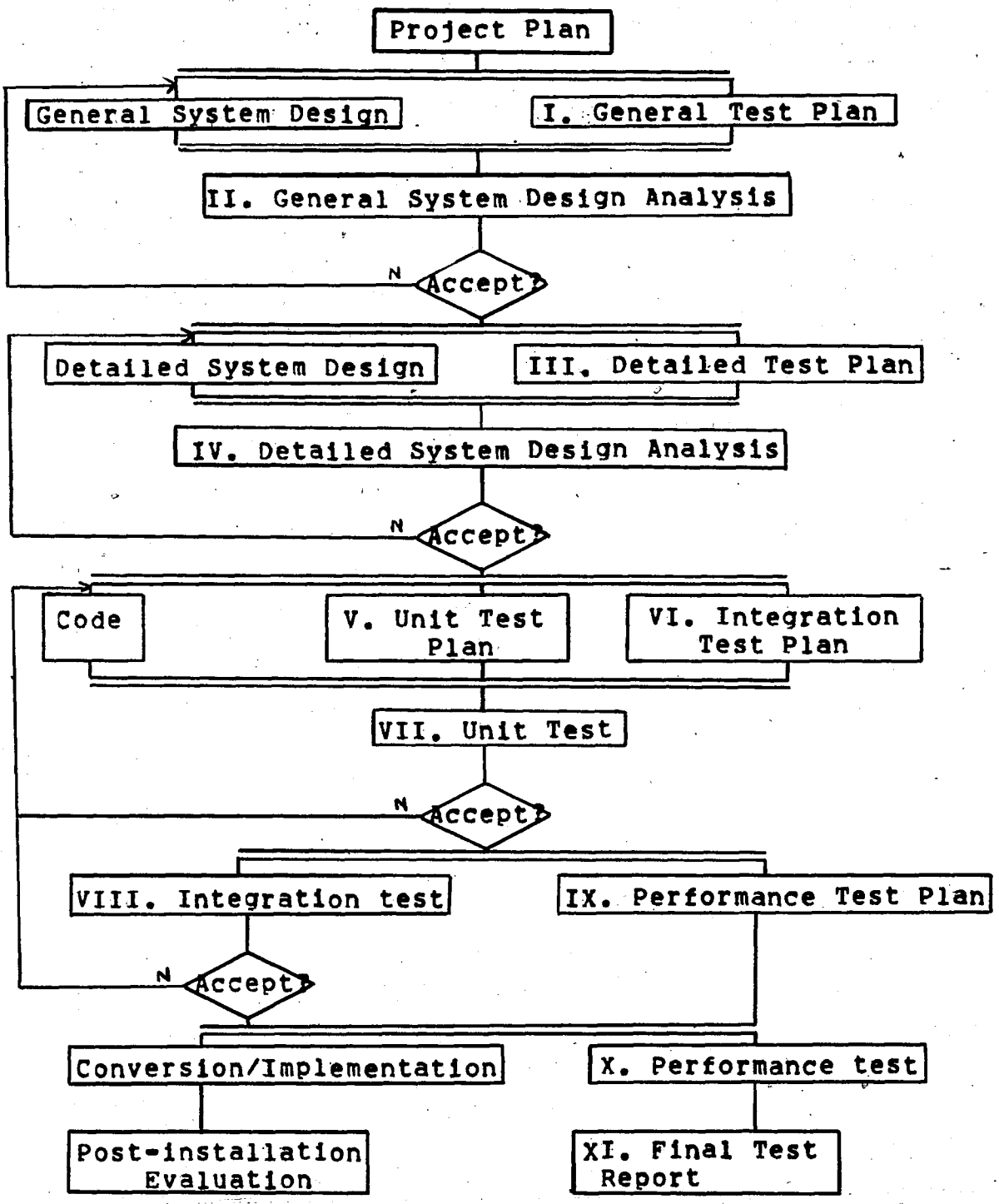
XI. F I N A L T E S T R E P O R T

SYSTEM TESTING PHASES
Figure 1

this perspective, and how the eleven phases of testing fit in.

This view is not unlike that of the system development process itself, which conceives of a system from a functional perspective, goes through finer and finer detail of design and coding, and finally constructs a finished product from its parts. Testing should be made integral to that process from start to finish. Normally, testing is considered necessary only after coding has been done, and is seen as a single step in the system development process. Since most errors are introduced in design stages, waiting until the task of coding and debugging is completed before examining the system's design is not only backwards, but counter-productive as well. Consequently, this methodology has been designed to parallel the system development process, at times even becoming a critical step before development should proceed. A general flow diagram of the way the test processes combine with the system development processes is shown in Figure 2.

A general outline of the testing methodology is set forth in this present chapter; specific guidelines and methods for implementing the various steps are then described in Chapters III and IV. Basically, the



INTEGRATED SYSTEM DEVELOPMENT/SYSTEM TESTING FLOW DIAGRAM
Figure 2

methodology proposed involves the use of a trained, permanent test team, somewhat along the same lines as a chief programmer team, with the chief tester heading the team and performing all major work (this will be more fully described in Chapter III); the extensive use of planning documents and specific guidelines and procedures for reporting results; the development or installation of automated test tools when possible and feasible; and finally, the use of backtracking and looping techniques. (Backtracking involves examining each design document against those which preceded it. Looping means that the development process cannot proceed to the next step until the previous step has been formally accepted by the testing team; design documents can be sent back to the "drawing board" as often as is necessary.) Another major issue addressed is that of establishing acceptance criteria, i.e. knowing when to stop testing (this is also discussed in Chapter III).

Implementation of some of the steps in the phases, such as performing static and dynamic analysis and test data selection, will be more fully explained in Chapter III.

At the end of this chapter, two charts summarizing the phases of the proposed methodology are included (Figures 3 and 4). The reader is invited to refer to these charts before, during, and after reading the accompanying text to maintain an overall view of the methodology.

B. Steps in testing

Preliminary Phase:

If the proposed methodology is being implemented for the first time within an organization, two preparatory steps must be taken. First and foremost, a testing team will be assigned and a chief tester appointed. Chapter III explains the nature of the test team in detail, with regard to training, experience, organizational structure, job descriptions, etc.

The first task assigned to the newly formed test team will be to conduct a feasibility study on software packages available to aid in program and system testing. This will help both in familiarizing the team with testing materials, and in developing a test environment which is general enough to be used in any future projects.

Once these steps have been taken, the following eleven phases can be used as the standard for any system development project.

Phase I. Develop General Test Plan

The project manager will be responsible for the development of a plan which will guide the overall test process for that particular project. Included in this plan will be:

a. Establishment of a general test schedule, especially with regard to the general system development schedule, and setting dates for completion of major milestones.

b. Allocation of resources and/or budget to the testing process for materials procurement (i.e. special equipment, reporting forms, etc.), as part of the total project resources/budget.

c. Establishment of guidelines for general acceptance criteria.

d. Specification of major reports which must be submitted to management to signal the completion of each significant

activity, such as the General System Design Analysis Report, Detailed System Design Analysis Report, etc.

In addition to providing general guidelines for the test team to follow, the purpose of this plan will be to ensure that certain criteria are met along the way.

The chief tester will receive a copy of this plan.

Phase II. General System Design Analysis

When the general system design has been completed, it will be submitted to the entire test team for review and analysis. Comparisons will be made between that and the system requirements documents to determine if the General System Design meets the user and/or system requirements as initially set forth. The General System Design will also be examined within itself for such qualities as feasibility, testability, etc.

The documents will first be examined by team members individually, and then the team will meet formally to compare notes. From that meeting the chief tester will compile the General System Design Analysis Report, which will be presented to the project manager and lead system analyst. This report will either recommend modifications to

the design or formally accept it. If modifications are recommended, the document is returned to the group responsible for that design. The changes must again be reviewed and approved by the test team.

A second reason for having the entire test team review the General System Design is to familiarize the testers with the overall scope of the system and with user requirements so that future test efforts are better understood. It is essential that the testers come away from this with a good understanding of the system's intentions, so that when they study the system in operation at a later phase, they can do so with a clear perception.

Phase III. Develop Detailed Test Plan

While the detailed design documents are being drawn up, the chief tester will begin work on the Detailed Test Plan. The purpose of this plan is to divide the General System Design into its functional components, which will be assigned to individual members of the test team. An individual will be responsible for testing a particular component until it has been integrated into the system. In addition, this plan will specify an overall methodology for code testing, which will include static analysis, test data

generation, and dynamic analysis techniques. If a testing aid software package is to be used or developed, that will be so stated here, along with a means for the procurement or development of the package.

This plan will include an analysis of the functional components themselves to determine the point at which each will be developed and implemented and the way in which it is to be integrated into the system (i.e. which elements must precede it, which must follow it, etc.). Also, for each component, any special conditions or constraints, data criteria, hardware/software requirements, and acceptance criteria will be stated. All this must be taken into account in the decision of how and when a component is to be tested.

Finally, the Detailed Test Plan will set up a projected schedule for detailed system design analysis, unit testing, and integration testing within the framework of the schedule given to the testing team by the project manager. The milestone dates will state which reports will be expected by particular dates (and from whom).

When the plan is finished, a copy will be sent to the project manager for approval. After being approved, copies will be distributed to the entire testing team.

Phase IV. Detailed System Design Analysis

Once the Detailed System Design has been completed, it will be submitted to the test team for analysis. The purpose of this analysis is to determine if the Detailed System Design is consistent with the General System Design, and if all parts of the Detailed System Design are accurate, feasible, implementable, testable, etc. In addition, it will provide testers with enough information so that they can begin preparation for unit testing.

Phase IV will proceed in the following manner:

1. The design components (or elements) will be divided up among the test team, as specified in the Detailed Test Plan. Each person will examine those designated design elements, comparing them to the earlier design documents for consistency and completeness, and analyzing them for accuracy, feasibility, etc. Results of each analysis will be summarized in a report to the chief tester.

2. At the same time, the chief tester will analyze the design element interfaces to determine how and where the various parts fit together, and any possible inconsistencies in those interfaces.

3. When each element has been examined separately, the test team will meet to perform a structured walkthrough of the logic. Results of the meeting will be summarized in a report by the chief tester.

4. From all of the reports submitted by the testers, the report of the structured walkthrough, and the report on design interfaces, the chief tester will write an executive summary, or Detailed System Design Analysis Report, which gives the status of the Detailed System Design (accepted, not accepted) along with recommended modifications.

5. This report will be submitted both to the project manager and to designated members of the design team for review. Those persons will meet within a week to discuss the report. This meeting should resolve any questions regarding modifications, and a result of the meeting will be a list of design modifications to be made. Any elements that may have been affected by the changes will be sent through Phase IV again, and the process repeated until all of the design elements meet the acceptance criteria.

Phase V. Develop Unit Test Plans

Coding should not begin until the design has been accepted. Concurrently, the test team will begin to plan for unit and integration testing. (Unit testing, as used here, refers to the testing of individual programs.) Each tester will be responsible for submitting a test plan for his/her assigned components to the chief tester, who is responsible for coordinating the process.

The testing methodology to be used will have been well-defined at this point, but the unit test plans will fill in any specific details which may be lacking in the Detailed Test Plan. Criteria for test data, and a plan for development of functionally-derived test data sets will be included in this document. The tester will state any tools or special materials, such as input/output stubs (dummy modules), which are necessary to test the unit, and will propose methods for their procurement or development.

The completed unit test plans will then be submitted to the chief tester, who will either accept the plan as is or recommend changes. Upon approval of each plan, the tester can begin development of test data sets, stubs, drivers, and any other tools which will become necessary during the unit testing phase.

Phase VI. Develop Integration Test Plan

While the testers are developing their Unit Test Plans, the chief tester is responsible for designing a plan for integration testing. This provides a scheme which defines a logical order for combining the individual modules, and insures that necessary drivers and stubs are available as needed.

The plan will state and diagram the precise order in which the modules will be integrated. It will also specify criteria for functional test data, means for test data development, stubs and drivers to be used at precise points and plans for their development, and acceptance criteria to be met before adding another module. Finally, it will present an approximate time schedule for module integration.

The plan will be submitted to the project manager for review and approval. Once it is accepted, the chief tester should begin assigning development of test data sets and driver and stub modules to members of the test team.

Phase VII. Unit Test

Unit testing begins when an individual program unit (the term "module" will be used to refer to a single program which is part of the system) has been coded and debugged, and appears to be performing satisfactorily. The purpose of unit testing is to determine if that particular module (in isolation) performs accurately, reliably, and in compliance with its functional specifications.

The programmer will provide the tester with a debugged program. Since the tester is already familiar with the intended functions and logic of the code, no detailed explanation or walkthrough with the programmer will be necessary at this be point, though possibly instructions on how to compile or execute the program may be needed. Upon receiving the code, the tester will first perform a static analysis to determine whether there are any suspicious constructs or data flow anomalies. Errors found at this point should be repaired by the programmers before any further unit testing is performed.

One product of the final static analysis will be a directed graph of program control flow, which will be used to derive additional test data sets to exercise the program's paths. Symbolic evaluation techniques (see

Chapter IV) will be used to determine path predicates, and an optimal test data set will be developed from the resulting values. These will be combined with functionally-derived sets to produce all of the test sets necessary to exercise the program.

The test data sets will be stored in a file or data base and dynamic analysis will be performed against each of the sets. Results will also be recorded and stored, and any deviations from expected output will be analyzed by the tester. Recommendations for revisions will be sent to the programmers. The entire Phase VII process will be repeated when the program is "fixed"; however, only those test cases which showed deviations (and any others which may have been affected by the change) need be retested.

When all the acceptance criteria for that unit has been met, or when the program has been fully exercised with all data sets producing correct output, the tester will certify the module as being acceptable, and submit a summary report to the chief tester.

VIII. Integration Test

Integration testing will begin when unit testing has been completed. The entire test team will work together to perform the integration, and the process will be overseen and guided by the chief tester.

The purpose of performing integration testing is to combine the separate modules in a logical fashion and to be able to pinpoint problem areas (especially with regard to module interfaces).

Top-down or bottom-up integration methods can be used. The highest or lowest level module will have the next level added to it (one module at a time), using all necessary stubs or drivers for calls, input/output, etc. When all data sets have been run and the results recorded, and the chief tester is assured that the two modules together are functioning correctly, the next module will be added. The process will continue until all modules have been successfully integrated. When an error is encountered, it will be returned to the programmer for "fixing" before proceeding with the integration process. Assuming the units had been performing correctly alone, the "bug" is most likely due to interface problems.

The end product of Phase VIII will be a system which performs accurately, reliably, and in accordance with the design specifications. The system is ready to be placed in a real environment and tested for other factors, as described below.

Phase IX. Develop Performance Test Plan

While system integration is taking place, the chief tester will begin work on planning for performance testing. What is meant here by performance testing is the analysis of those factors which can only be examined in a real environment. The general quality of the software logic has already been assured in earlier phases of testing; such elements as response time, back-up, recovery, user interface, reaction to stress and heavy volumes of data, etc. will now be considered. It will designate the performance factors to be tested, and specify the manner in which they will be tested, the desired results, and a schedule for testing each element.

Information for deciding which factors remain to be tested will be derived from the earlier system requirements documents. The Detailed Test Plan will also have stated which elements would not be testable until the system was

placed in an operational environment.

This testing plan could actually be drawn up any time after the Detailed Design Plan is approved. However, it is suggested that this be done just prior to the testing itself, as up until that time, schedules may have been so altered that the actual conversion schedule no longer matches the original plan. Also, additional factors for performance testing which were not considered earlier may become obvious during integration testing.

The completed plan will be submitted to the project manager for approval. When the plan is accepted with no further revisions, copies of the plan will be distributed to the test team, and assignments made to develop any tools or materials which will be needed to perform the tests.

Phase X. Performance Test

The entire test team will take part in performance testing, which will parallel the conversion/implementation phase and immediately follow the acceptance of the integrated system. As noted, the purpose of performance testing is to ensure that the system performs according to its original specifications in an operational environment. In addition, performance testing will be used to determine

tolerance levels of the system under stress and the reaction of the system when certain error conditions are forced, such as erroneous data or incorrect passwords. The reactions of the user to the system will also be studied at this point.

Any serious problems encountered will be immediately remedied; in many cases this can be done by modifying hardware configurations or physical facilities. Such items as passwords and erroneous data will have been checked earlier in Phases VII and VIII. However, since these errors can create serious problems, they will be retested in front of the user for added insurance.

Each type of test performed will be summarized in a report to the project manager (System Performance Report), covering both problems encountered and their remedies, system tolerance levels, causes of down time, system performance under various conditions, etc.

Phase XI. Submit Final Test Report

When the system has been turned over to the user, the chief tester will be responsible for summarizing the testing process and results. This will parallel the Post-Installation Evaluation phase of system development and will give the chief tester a chance to review not only the

testing process, but the methodology as well. Some items which will be included in the report are:

1. common errors which appeared in the design and how they can be avoided;

2. problems encountered and how their solution was developed;

3. problems which remain unsolved;

4. a comparison of the system as it was conceived and the system as it evolved, showing major deviations from the original intentions; and

5. an evaluation of the effectiveness of the testing methodology, and suggestions for its improvement in future projects.

The final report not only gives management ideas on how the system development process could be improved and what mistakes could be avoided, but also gives the chief tester a chance at self-evaluation and improvement. The report will be formally presented to management as part of the Post-Installation System Evaluation process.

This concludes the description of the eleven phases. Figure 3 summarizes the major points of each phase, and Figure 4 outlines the steps of the entire process in detail.

Phase	Description	Responsibility	Follows	Parallels	Precedes	Source Documents Needed	Products
I	Develop General Test Plan	Project Manager	Project plan	General System Design	General System Design Analysis	Project plan	General Test Plan
II	General System Design Analysis	Test Team	General System Design	---	Detailed System Design	General System Design Requirements Documents	General System Design Analysis Report
III	Develop Detailed Test Plan	Chief Tester	General System Design	Detailed System Design	Detailed System Design Analysis	General System Design General Test Plan	Detailed Test Plan
IV	Detailed System Design Analysis	Test Team	Detailed System Design	---	Coding	Detailed Test Plan General System Design	Detailed System Design Analysis Report
V	Develop Unit Test Plans	Test Team	Detailed Test Plan	Coding	Unit test	Detailed System Design Detailed Test Plan	Unit Test Plans (one for each module)
VI	Develop Integration Test	Chief Tester	Detailed System Design	Coding	Integration test	Detailed System Design Detailed Test Plan	Integration Test Plan
VII	Unit Test	Testers	Coding	---	Integration test	Code Unit Test Plans	Acceptance report of certified software
VIII	Integration Test	Test Team	Unit Test	---	Conversion/ Implementation	Integration Test Plan Tested Code	Acceptance report of integrated, certified system
IX	Develop Performance Test Plan	Chief Tester	Coding	Integration Test	Conversion/ Implementation	General System Design Detailed System Design	Performance Test Plan
X	Performance Test	Test Team	Integration Test	Conversion/ Imp.	Post-Installation Evaluation	Performance Test Plan	System Performance Report
XI	Submit Final Test Report	Chief Tester	Conversion/ Imp.	Post-Inst. Eval.	---	All test and design documents	Final Test Report

SUMMARY OF TESTING PHASES
Figure 3

I. Project manager develops General Test Plan
   - set up test team (if not permanent group)

II. General System Design, Requirements Documents submitted to test team
   - test team members examine documents individually
   - test team meets to discuss the documents
   - chief tester compiles General System Design Analysis Report
   - chief tester meets with project manager and lead system analyst to discuss findings
   - if modifications are in order:
      -- design group makes recommended changes
      -- Phase II is repeated
   - otherwise:
      -- proceed to Phase III

III. Chief tester develops Detailed Test Plan
   - copy of plan is submitted to project manager
   - if modifications are in order:
      -- return to Phase III
   - otherwise:
      -- distribute copies to test team
      -- proceed to Phase IV

IV. Detailed System Design submitted to test team
   - divide elements among team, according to Detailed Test Plan
   - tester studies design and submits a report of findings to the chief tester
   - chief tester analyzes design interfaces
   - test team meets to perform structured walkthrough
   - chief tester writes Detailed System Design Analysis Report
   - chief tester submits report to project manager and members of design team
   - chief tester meets with project manager and design group to discuss findings
   - if modifications are in order:
      -- design group makes recommended changes
      -- Phase IV is repeated
   - otherwise:
      -- proceed to Phase V

OUTLINE OF TESTING PROCEDURES
Figure 4

V. Testers develop Unit Test Plans
- plans are submitted to chief tester
- if modifications are in order:
  -- Phase V is repeated
- otherwise:
  -- testers begin developing necessary tools
  -- proceed to phase VI

VI. Chief tester develops Integration Test Plan
- plan is submitted to project manager
- if modifications are in order:
  -- Phase VI is repeated
- otherwise:
  -- proceed to Phase VII

VII. Code is submitted to tester responsible for that module
- static analysis is performed on code
- if errors are detected:
  -- code is returned to programmers for "fixes"
- directed graph of program control flow is generated
- symbolic analysis is performed on graph
- optimal set of test cases is produced
- dynamic analysis is performed using the test cases
- execution history is recorded in data base
- if errors are detected:
  -- code is returned to programmers for "fixes"
  -- Phase VII is repeated
- otherwise:
  -- Acceptance Report is submitted to chief tester
  -- proceed to Phase VIII

VIII. Begin Integration Test
- add in next level module
- if errors are detected:
  -- code is returned to programmers for "fixes"
  -- continue testing with same module
- otherwise:
  -- continue Phase VIII until all modules have been added

OUTLINE OF TESTING PROCEDURES
Figure 4

IX. Chief tester develops Performance Test Plan
- copy of plan is submitted to project manager
- if modifications are in order:
  -- Phase IX is repeated
- otherwise:
  -- copies are distributed to test team
  -- proceed to Phase X

X. Begin Performance Test
- if errors are detected:
  -- correct problem areas
- otherwise:
  -- develop System Performance Report

XI. Chief tester writes Final Test Report
- report is presented to management

OUTLINE OF TESTING PROCEDURES
Figure 4

Chapter III. Managing the testing process

A general testing methodology has been presented, but little has been said thus far regarding the details of its actual implementation. As with any system development process, there are two perspectives to consider. From the one side, management is responsible for organizing the staff, delegating responsibilities, providing necessary resources, and determining when a process can be concluded. On the other side is the technical viewpoint, where the delegated persons with the necessary expertise and resources actually carry out the details of the plan. But the two sides are not independent. Unless management has done its job well, the technical people will lack the necessary time, training, and resources to do so.

This chapter addresses decisions which management must make, especially prior to and during the initial implementation of the testing methodology. Three areas are discussed, including:

A. Assignment of testing activities: The methodology described in Chapter II proposes the establishment of a chief tester team prior to its implementation. The present section examines various organizational means of assigning

testing activities, and explains why the chief tester team was chosen as the best approach. The proposed organizational structure of the test team is then discussed. Finally, at the individual level, the training and responsibilities of a tester are described.

B. — Allocation of resources: Included in the General Test Plan (Phase I) should be a specification of the time and budget to be devoted to the testing process. Typical time percentage allocations for various steps in a conventional system development process are compared with allocations for those of the proposed methodology to demonstrate how the eleven phases can be combined with traditional procedures. Recommendations are set forth for scheduling and budgeting.

C. Criteria for acceptance: Each of the planning phases (Phase I, Phase II, Phase V, and Phase VI) involves an establishment of acceptance criteria, i.e. means of determining when a system and/or program has been thoroughly tested and judged acceptable. This section discusses various recommended ways of determining those criteria, which will vary slightly for each of the different phases.

A. Assignment of testing activities

Miller [1978] has identified several common negative attitudes maintained towards the testing process:

1. Testing is a dirty business.

2. Testing is unimaginative and lacks a sense of adventure.

3. Testing is a menial task.

4. Testing is a difficult task due to the complexity of programs and a deficient technology.

5. Testing is not important, and involves too much work in too little time.

6. Existing tools are not well-developed.

7. There is too much ad hoc methodology and too few generally accepted principles.

8. Testing has a negative reward structure, requiring a "critic's mentality".

Considering these points, it is no wonder then that testing is hardly ever done well, if at all.

Miller also proposes some possible ways of inspiring those responsible for testing, including:

1. Presenting testing as an especially challenging activity which requires creativity.

2. Stressing the importance of testing in creating a more reliable product.

3. Pointing out the additional opportunities for creative innovation, since the technology is neither well understood nor well developed.

These arguments may be partially convincing to some, but will probably not serve to inspire programmers or analysts who are already overworked.

One important first step the project manager must take is to consider who to assign to the testing activities. The methodology described in Chapter II proposes the establishment of a permanent test team, which is independent from a programming or design team. Several alternative approaches were considered in arriving at this decision.

1. Testing assignment alternatives

(a) The testing activities can be delegated to the analysts and programmers, with each person responsible for testing his or her own designs or programs. This is not recommended unless there is a critical shortage of personnel, since it is difficult to detect errors in one's own work. Programmers would rather prove that their work performs intended functions than assure that it does not perform unintended functions. Moreover, they cannot always envision abnormal cases, since their prime concern is making the code work with normal situations.

(b) In a second approach, testing assignments can be divided among the same individuals, but in a way that no one person is responsible for checking his/her own section. The segments of design would be reassigned among the design team, and likewise the program modules reassigned among the various programmers for testing. This is not satisfactory for reasons which will be discussed later.

(c) A third approach is to allow the programming team to test the design, and to have the analysts test the program(s). With this method, fresh ideas could be obtained from the different groups, and all members could be involved in the various aspects of the total system.

These last two approaches offer at least partially suitable solutions, but there are still problems. For one, analysts and programmers may not be adequately prepared to understand or implement testing theory. In addition, in either approach, individuals would be evaluating peer performance (and in some cases a supervisor's performance), and might consequently be less critical. Finally, the negative attitudes previously cited are apt to occur in these approaches, too.

(d) A fourth alternative is to assign the entire testing process to a team of programmers and/or analysts on a rotating basis. For each project, a new team is assigned, whose only task is to test that particular system. This approach is somewhat better than those in which designers and programmers test either their own or their peers' work, but some experience would be lost each time a new group is assigned. Also, the negative attitudes are likely to appear here as well.

(e) The final and recommended approach is for the test team to be appointed on a permanent basis. This emerges as the best solution for several reasons. The members of the test team will have the time to learn their jobs well and can improve techniques from project to project. They can

concentrate on one task, and thus perform it better.

Establishing testing as a separate activity makes it a more important part of the process, equal with design and programming. It also relieves analysts and programmers of the (to them) tedious chore of testing. A programmer will provide the tester with code which is debugged and operating well with normal data. However, the tester is trained to know how to select data which will reveal errors, and will be more apt to find errors than the person who wrote the program. One might fear that a programmer will be more careless if someone else has to worry about performing the tests. However, the programmer will get an incorrect program back to be repaired, and would rather do the job right the first time than have the code rejected. This would suggest that, if anything, he/she will probably produce better code, knowing that careless errors will only have to be mended later.

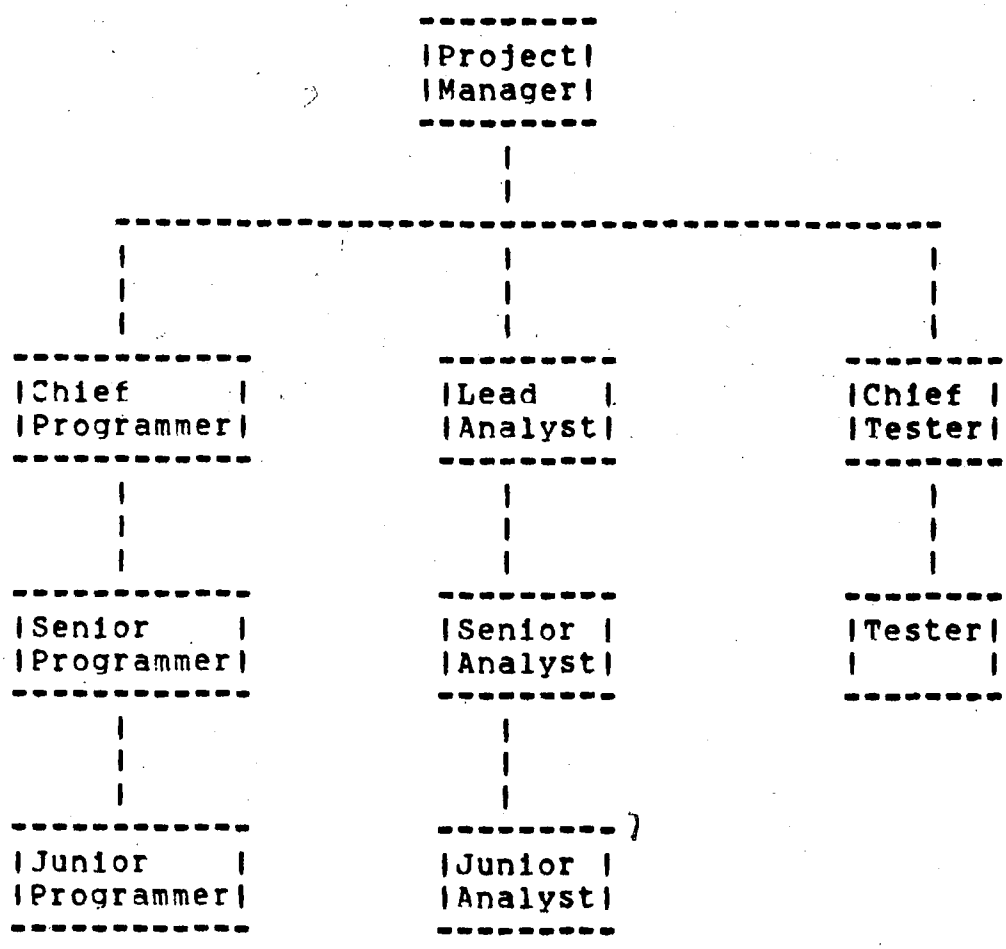
2. Organizing the test team

In the proposed methodology, the test team leader is the person responsible for detailed reporting, planning, and supervising the test process. That person must be extremely competent and knowledgeable in both testing theory and techniques, and must also be able to organize ideas and to

supervise people. For these reasons, the test team should be structured as a chief tester team.

The chief tester team is based on the concept of a chief programmer team as developed by IBM [Baker,1972]. As with the chief programmer team concept, it recognizes different levels of competence among testers, and establishes the chief tester as the person responsible for overall test planning and procedures. The chief tester will be the most creative and productive team member. If possible, there should be a backup to the chief tester, responsible for working with and filling in details assigned by the chief tester, and for taking over in case the chief tester should leave. Junior testers assigned to the team will be largely responsible for implementing the assigned details of the test plan.

Figure 5 depicts an organizational approach recommended in assigning the test team. The chief tester is on the same level as the chief programmer or lead system analyst, and works informally with both. A tester has informal lines of communication with both senior and junior programmers, and is on the same organizational level as senior programmers and/or analysts.



ORGANIZATIONAL CHART
Figure 5

Responsibilities of the test team have already been generally described in Chapter II. In terms of specific tasks, they include:

Chief tester responsibilities:

1. Oversee the test process and supervise the test team;
2. Present all major reports to management, designers, etc.;
3. Develop a Detailed Test Plan;
4. Establish specific schedules;
5. Establish specific test acceptance criteria;
6. Decide on testing methodologies, software test packages, etc.;
7. Examine design interfaces;
8. Serve as a liason between designers and testers;
9. Plan for module integrations;
10. Plan for performance tests;

11. Review and summarize the work of the testers (i.e. the results of all tests).

Tester responsibilities:

1. Examine and analyze General System Design;

2. Examine and analyze assigned parts of Detailed System Design;

3. Develop unit test plans for assigned design elements;

4. Develop stubs, drivers, or test software necessary to perform unit tests and/or integration tests;

5. Develop and maintain test cases for unit and integration tests;

6. Perform unit tests;

7. Perform integration tests;

8. Perform system performance tests;

9. Interact with programmers and coders to explain where and how the code should be fixed;

10. Record all findings and provide the chief tester with required reports.

In addition to the test team, the project manager will have specific responsibilities with regard to the testing process, including:

1. Develop a General Test Plan;
2. Formulate a test budget;
3. Establish a general test schedule;
4. Establish general acceptance criteria guidelines;
5. Supervise the chief tester;
6. Review and approve all major reports submitted by the chief tester.

Since a test team will be appointed on a permanent basis, there will be a need for testers within the organization. A tester will be a person whose job it is to oversee the system development process from a critic's viewpoint, who has been specifically trained in the art of testing and who sees it as an essential part of the process.

3. Testing as a career

A tester's training will be similar to that of a system analyst or high level programmer, with additional training in testing theories and techniques (such as those discussed in Chapter IV). Unfortunately, few (if any) universities or colleges offer courses in system testing. One recourse might be to locate some of the testing experts in the country, and hire one as a consultant either to help establish an in-house training course or to personally train one or two persons within the company. If a large university is located nearby, especially if some rapport has already been established with that university, another possibility might be to contact the chairmen of appropriate departments to see if a course in system testing could be arranged. The in-house training course is a more feasible approach, however, since the university may not be willing to develop a course for one or two people; also, an in-house course can be more tailored to the needs of the organization. It can be developed and revised by the chief tester over a period of a few years, and should include:

a. A set of printed material on testing theory;

b. A sample set of programs and design documents with known errors, and some type of self-paced guide to discovering those errors;

c. On-the-job training using a mentor system.

More experience should be demanded for testers than is for programmers or junior analysts. A tester will have to have worked as either a programmer or junior analyst first, and should have a good understanding of the system development process.

Grades of testing expertise should be established (i.e. junior tester, senior tester), and a high level, or senior tester should be on the same level as a supervisor of a programming or design team in terms of both salary and prestige. The next step up from a "senior" or "chief" tester should be project manager, division head, or whatever similar position there may be within the corporation.

Testers will perform better when they are working with a clear methodology, and are responsible for performing certain tasks in some logical order and for generating certain reports at specific times. Providing testers with automated aids will help improve their productivity, and make the job less of a chore. Establishing these positions

as ones with required skills, background, and training, with opportunities for advancement, will aid in creating an effective test team. Perhaps even changing the job title to something more euphemistic, such as "Quality Assurance Engineer", might contribute to changing current negative feelings toward testing.

4. Initial test team creation

It is not likely that a company wishing to implement this methodology will already have a test team. A necessary first step is therefore to locate that person within the department (or hire a new one, if necessary) who has had experience with testing and/or has shown a great deal of ingenuity and accuracy as either a senior programmer or senior analyst, and to appoint that person as a chief tester. This should be done at least six months before the methodology will be put into effect. The chief tester will ordinarily spend this time learning as much as possible about system and program testing, attending seminars, workshops, and conferences, viewing demonstrations and gathering materials on testing packages, and spending time with knowledgeable consultants. One of the early responsibilities of the chief tester will be to establish a prototype training course for the first members of the test

team. (This course will, after necessary revisions, become the regular training course.)

No less than two months prior to the implementation of the testing methodology, a test team should be appointed. The chief tester will be responsible both for preparing the testers in necessary skills and for introducing them to the testing methodology. This will include literature, discussions, and possibly visits to sites using some sort of testing program. As part of this training, the test team will work together in examining and evaluating test tools which are available for purchase. The results of this analysis will be a feasibility study which will recommend purchase of various tools and/or in-house development of similar tools.

The team will remain together on a permanent basis. Initially, they will implement the testing methodology on a single project. In a large shop, they could later be assigned to a number of projects, or the team could split and new teams be formed, drawing on the expertise of the charter team members.

B. Allocation of resources

The proposed methodology inserts the various testing phases into the system development process, implying that some accomodation must be made for them in terms of time and budget. Since projects seldom finish within time or budget restrictions, this implies a longer system development period and/or a larger budget. Lengthening the development process or increasing the already high cost of software to include a formal testing process could sound terribly unattractive to both the customer and the developer.

However, one need only consider the advantages in doing so to be convinced that it is necessary. For one, the cost of maintaining or "fixing" the system should be substantially reduced, since it should perform more according to specifications and needs at release time. Currently, designers are preoccupied with getting the system out to the customer as quickly as possible, and then worrying about "fixing" the system to perform according to need. One need only see that maintenance accounts for anywhere from 37% [Alberts,1976] to 70% [Fairley,1978] of a system's life cycle to make this conclusion. Customers will be more satisfied with a product that performs the proper functions reliably, and are more likely to return for future

needs. In addition, no potential customer operates in a vaccuum; other users of a system are likely to be queried to determine its reliability. An attractive system which requires little maintenance is likely to increase requests for installations at similar sites, or for new developments. If the developers are operating within a corporation, the image of that department is likely to improve. Finally, since testing should be totally integrated into the development process, its costs will tend to be invisible to the consumer.

Figure 6 demonstrates how testing can be integrated into the system process. The column on the left represents a typical current methodology, and that on the right the proposed integrated methodology. The figures on the left are approximate, and have been constructed merely for contrast. Those on the right demonstrate how the proportion of time spent on each activity might be decreased in the overall time frame to allow for testing the product of that activity. (Absolute time, however, will remain constant.)

In the integrated method, 35% of the development time has been allocated to testing, representing a 35% increase in overall time for development. Of that total, 7.5% has been allocated to testing prior to coding. Although this is

Current Method	% Time
I. General System Design	15%
II. Detailed System Design	25%
III. Coding	20%
IV. Unit test	8%
V. Integration test	7%
VI. Conversion and Implementation	20%
VII. Post implement. evaluation	5%

Integrated Method	% Time
I. General System Design	10%
II. General System Design Analysis	2.5%
III. Detailed System Design	20%
IV. Detailed System Design Analysis	5%
V. Coding	15%
VI. Unit test	10.5%
VII. Integration test	12%
VIII. Conversion and implementation	15%
IX. Performance test	5%
X. Post implementation evaluation	5%

ALLOCATION OF TIME TO TESTING
Figure 6

unusual, studies have shown that up to 80% of all errors are introduced during the design phase [Alberts,1976]. At least one-quarter of the time spent on design should be allocated to examining that design for accuracy and adherence to system specifications; if it takes a year to design a system, that design should be inspected by the testing team for no less than three months. Also, time spent in unit and integration testing has been substantially increased.

In practice, the development time will be increased accordingly (as will the amount spent), so that the entire development process will be longer. If the current methodology takes 1000 man-hours for development, the proposed methodology will possibly take 1350 man-hours. However, since the cost of maintaining the system should be substantially reduced, the cost of the system's life cycle might be lower than had it been developed the conventional way. Overall time and effort expended in producing a satisfactory system might also be lessened if the post-installation maintenance period decreases as expected.

With regard to expenditures, implementing the methodology will require some initial investments, e.g., consultant, software packages, etc. It necessitates the use of automated aids and understanding of test theory in order to be effective, and especially in early stages, the chief tester should be provided with as many resources as are necessary to do the job well. These are one-time costs, however, and eventually will be to the company's advantage. Operational costs will include salaries, training programs, worksheets and other supplies, etc.

Within the General Test Plan (Phase I), the project manager must establish schedules for testing. Since this is done after the completion of the project plan, the general schedule of the project can be used as a framework to determine major test milestones and final dates for completion of test reports. Budget allocations for specific projects will mainly be for salaries and materials. A single test team of approximately five persons should be sufficient for a project, provided they are given enough time and resources to complete their tasks. As they become more experienced and proficient, the time requirements can be expected to decrease.

For the first implementation of the methodology, the figures presented in Figure 6 can be proportionately fit into the organization's current methodology. With each new plan, they should be revised and adjusted as more is learned from previous projects.

C. Criteria for acceptance

Unlike programming and design, testing is a negative process. A programmer declares the code finished when it produces desired results. The tester's job is more difficult, for the tester must assure that the code always produces desired results, and never produces undesirable results. Since "always" and "never" are difficult, if not impossible, to prove in reality, the tester must have criteria to know when a system and/or program has been tested "enough".

Each of the test plans (i.e. General Test Plan, Detailed Test Plan, etc.) described in Chapter II features the inclusion of acceptance criteria. Criteria must be established in advance to determine both when to accept the product and/or when to conclude testing. Some methods commonly used, e.g., to stop testing when the scheduled time expires or when all test cases execute without detecting

errors, have proven inadequate.

A better approach is to define completion in terms of specific test case design methods. For example, the criterion for test completion can be detection of some predefined number of errors, derivable from: (1) an estimate of the total number of errors in the program (avg. errors per line x no. of lines), (2) the percent that can be detected through testing, and (3) the fraction that may have originated in a particular design process. These values can be derived from previous experience. By plotting the number of errors found per unit of time, the testing process can end when the plot levels off to an acceptable minimum; at this point, further testing may result in diminishing returns.

Another method would be to determine the number of tests required to attain a certain predefined percentage of coverage, say 90%. If test cases are designed to exercise a specific portion of the software, there are ways of determining how many test cases are necessary to cover most of the system. (This will be more fully discussed in Chapter IV.) In the above method, the figures used are statistically derived and are not convincingly reliable. Determining acceptance criteria from percentage of program coverage provides a firmer, more reliable figure, which

makes this method somewhat better. Chapter II recommends defining a percentage of coverage for program testing, although the error estimate method can be used for additional quality assurance.

Yet another approach is to establish a system of metrics which can be used to quantitatively evaluate the quality of the software [Boehm,1976]. Under this method, the following steps can be taken:

1. Identify desirable characteristics of the software, such as understandability, completeness, conciseness, consistency, maintainability, reliability, etc.

2. Develop a set of metrics, or rating scale associated with each of the characteristics.

3. Determine to what extent the actual software meets the evaluation criteria metrics.

This method will best apply to overall quality assurance, and is recommended in establishing criteria for the General and Detailed Test Plans.

Within the General Test Plan, the project manager must establish general criteria for system acceptance. Using the metric approach, the prominent general requirements are identified and a weighted value assigned to each. For example, a password security requirement might be rated higher than a response time requirement. A total of quality points can be ascertained, and criteria can be established by declaring that the system must attain a minimal number of quality points before its acceptance. For example, if the sum total is 150, the system may be acceptable when it has acquired 140 points.

When formulating the Detailed Test Plan, which will refine these criteria, the chief tester should look at the components of each requirement separately and determine how many quality points each component must achieve before being acceptable. For example, if password security has been rated 9 on a scale of 10, and if there are several elements of the detailed design which contribute to password security, the Detailed Test Plan should be specific as to how many points each component is worth. The chief tester should also state what that rating means in terms of specific tests, and how that figure is to be derived. It might mean, for example, a 90% path coverage of a program module. The sum total of all components of a requirement

should equal the value previously assigned to that requirement in the General Test Plan.

The Detailed Test Plan will provide the acceptance criteria guidelines for the Unit Test Plans. If a module is expected to be exercised with 90% coverage of all possible paths, the Unit Test Plans should design test cases intended to do so, and unit testing acceptance criteria should be based on those carefully selected test cases.

Establishing acceptance criteria is perhaps one of the most difficult tasks facing a program manager. It is essentially a matter of deciding how reliable a system must be. This topic alone readily lends itself to full-scale research, and the preceding paragraphs have merely skimmed the surface. Some recommended techniques have been sketched, but it is up to the project manager and chief tester to define the criteria more precisely.

This concludes the discussion of managerial guidelines for testing. This chapter has suggested ways of implementing the methodology in a traditional system development shop. The next chapter deals with specifics of the proposed methodology, demonstrating how established test theory is to be put into practice by the test team.

Chapter IV. System Testing Tools and Techniques

In order to implement the proposed testing methodology, the testing team should have a good set of working tools, and a knowledge of various testing techniques. "Tools" is used here loosely as meaning any software aids or documentation forms, or any practical means of implementing theory and/or making the testing process simpler and more standard. Whether these tools be manual or automated, it is important that they be usable, reliable, and available when they are needed.

This chapter steps through the proposed methodology, suggesting tools, documentation forms, and techniques which can be used in its implementation. The phases of the testing methodology addressed here are the testing rather than the planning activities. They include:

Phase II: General System Design Analysis

Phase IV: Detailed System Design Analysis

Phase VII: Unit Test

Phase VIII: Integration Test

Phase X: Performance Test

Sample documentation forms and charts are given, using the design documents of Appendix A for illustration. These design documents are not meant to be complete, but merely to aid in demonstration.

At the end of this chapter, a chart summarizing recommended techniques is provided (Figure 20).

A. General System Design Analysis (Phase II)

Phase II of the testing methodology requires analysis of the General System Design by the test team. This is done to insure that the General System Design matches the client's needs.

The first step is to determine whether for each of the customer's expressed requirements, the General System Design shows a corresponding feature. By cross-referencing those customer requirements which correspond to elements of the general design, it can be determined which, if any, of the customer requirements are missing from the general design, which elements of the general design are superfluous, and which elements of the design overlap. A chart similar to Figure 7 (Customer Requirements/General System Design Cross

CUSTOMER REQUIREMENT	GENERAL SYSTEM DESIGN ELEMENT
1. Automatic billing	3.1
2. Customer history	1.1
3. Flag delinquent customers	2.2
4. Itemized receipt	none
5. Report on items missing	3.2
6. Report of items requiring special attention	3.3
:	:
:	:
:	:

Summary:

Missing Requirements: 5. Itemized receipt

Superfluous Requirements: None

Overlapping Requirements: None

CUSTOMER REQUIREMENTS / GENERAL SYSTEM DESIGN
CROSS REFERENCE
Figure 7

Reference) should be used to perform this analysis.

The next step is to examine the design elements within the framework of the General System Design. It should be determined whether each component is a functional requirement or a performance requirement. Functional requirements include the types of reports to be generated, means of update, etc., while performance requirements include response time, maximum acceptable down time, etc. An individual element should be analyzed to determine:

(1) Its feasibility within technological/resource limitations: for example, is two-second response time feasible given equipment restrictions?

(2) Its completeness: can the requirement be implemented as described, or is more information necessary before proceeding to detailed design?

(3) Its consistency: Are there design elements which contradict other design elements? Are hardware suggestions incompatible or inconsistent with performance requirements?

(4) Its testability: Will it be possible to test a certain requirement? If so, at which stage should it be done and what criteria should be met? For example, response time cannot be tested until the system is intact in an

operational environment.

(5) Its necessity: Is the requirement really necessary to implement the system?

An example of a document which can be used to display the results of the analysis is depicted in Figure 8, the General System Design Evaluation Form.

Each tester, after studying both the General System Design and the system requirements documents, will individually fill out these (or similar) charts before the group meeting. When the test team meets to discuss the General System Design, all of the results should be compared.

B. Detailed System Design Analysis (Phase IV)

The Detailed System Design documents are analyzed in Phase IV of the testing methodology. During this phase, three major activities will be completed:

(1) The testers will examine parts of design documents separately.

ELEMENT	FEASIBLE	COMPLETE	CONSISTENT	TESTABLE	NECESSARY
1.1	y	n	y	y	y
1.2	y	y	y	y	n
2.1	y	y	y	y	y
2.2	y	n	y	y	y
2.3	y	y	y	y	y
2.4	y	y	y	y	y
2.5	y	y	y	y	y
3.1	y	y	y	y	y
:	:	:	:	:	:
:	:	:	:	:	:
:	:	:	:	:	:

Summary:

1.1 The customer information should include customer name

1.2 Price should not be kept with item information

2.2 "Delinquent" should be defined (i.e. A customer is delinquent if ...)

GENERAL SYSTEM DESIGN EVALUATION FORM
Figure 8

(2) The chief tester will analyze design element interfaces.

(3) The test team will meet to discuss the results of (1) and (2), and to perform a structured walkthrough of the design logic.

The purpose of this analysis is to insure that the design documents are complete, consistent, and feasible. The Detailed System Design documents should not only be examined in themselves, but also against General System Design documents.

As with the General System Design, there should be a Detailed System Design element to match each section of the General System Design. This can be assured using a chart similar to Figure 7. This analysis will be made while the design components are being divided among the test team, first at a general level by the chief tester, then more specifically by each tester with his/her own section. Figure 9 (General System Design/Detailed System Design Cross Reference) illustrates the use of such a chart.

After the Detailed System Design has been divided up among the test team, it will undergo careful scrutiny. As with the General System Design, the Detailed System Design

GENERAL SYSTEM DESIGN	DETAILED SYSTEM DESIGN
1.0. Data base	1.0. Data base design
1.1. Customer information	1.1.c. Customer record 1.3. Data elements
1.2. Item information	1.1.b. Item record 1.3. Data elements
2. On-line functions	2.0. On-line functions
2.1. Search by name, account no., or item ID	2.5.1. Customer look-up 1.4.c. Data base search by item ID : :
2.5. Item entry	2.5. Item entry
2.5. Item identifier slip	none
:	:
:	:
:	:

Summary:

Missing Design Elements : 2.1. Entry to data base by customer account number

2.5. Item identifier slip

Superfluous Design Elements : 2.5.1.c(2) Automatic request for charge account

Overlapping Design Elements : None

GENERAL SYSTEM DESIGN / DETAILED SYSTEM DESIGN
CROSS REFERENCE
Figure 9

should be analyzed to determine:

(1) Feasibility, given technological/resource limitations. If a particular design is discovered to be impractical or unfeasible at this point while its corresponding general specification had been judged feasible, it is possible that the design does not match the specification, so that it needs reworking. It is also possible, of course, that the feasibility of original specification was not thoroughly considered; in such a case, some reworking of the general design would be necessary.

(2) Completeness. The design should be detailed enough for the programmer to be able to code, and should therefore be as unambiguous as possible. Any areas which do not have enough information should be noted.

(3) Consistency. Especially if different people have designed different portions of the system, it is possible that incompatibilities exist between the various parts. Each design element should be examined to determine if it is consistent with the rest of the system. An example of an inconsistency of this sort would be a proposed file structure which cannot be implemented on the proposed hardware/software combination.

(4) Testability. This is not an absolute necessity, but it should be noted at this point if a design element cannot be tested. This will be useful information when it comes time to test the coding.

(5) Necessity. The design may have overlapping elements, which could result in some degree of superfluity. This might have resulted from two or more designers working on separate components, and proposing redundant file structures, programs, etc.

The results of this analysis can be displayed on a form similar to Figure 10, the Detailed System Design Evaluation form.

In addition to this evaluation, any equations used in the design should be checked against standard references to determine their correctness. Design algorithms can be checked by manually executing them using sample data.

While the testers are examining the design elements for the aforementioned items, the chief tester analyzes them for interface consistencies. The results of this analysis can be recorded in a chart such as that of Figure 11, or Design Consistency Check, which is useful as a checklist in

DESIGN ELEMENT	FEASIBLE	COMPLETE	CONSISTENT	TESTABLE	NECESSARY
1.1.a.	y	y	y	y	y
1.1.b.	y	y	y	y	y
1.1.c.	y	y	y	y	y
1.1.d.	y	y	y	y	y
1.2.a.	y	y	y	y	y
1.2.b.	y	y	y	y	y
1.2.c.	y	y	y	y	y
:	:	:	:	:	:
1.4.a.	y	y	y	y	y
:	:	:	:	:	:
2.5.1.b (2)	y	n	y	y	y
2.5.1.c (2)	y	y	y	y	n
:	:	:	:	:	:
:	:	:	:	:	:
:	:	:	:	:	:

Summary:

2.5.1.b(2) The design does not explain clearly what will happen after a delinquent customer message appears.

2.5.1.c(2) It is not necessary to query a new customer to open a charge account.

DETAILED SYSTEM DESIGN EVALUATION FORM
Figure 10

DESIGN ELEMENT	DEPENDENT ELEMENT ID	DEPENDENT ELEMENT INTERFACE	DEPENDED UPON ELEMENT ID	DEPENDED UPON ELEMENT INTERFACE
2.5.1.a	2.5.1b	Customer-name	2.0	Call from supervisor
	2.5.1c	Customer-name	---	---
	2.5.2	Customer-name	---	---
2.5.1.b(1)	2.5.1b (3)	(display)	2.5.1.a	Customer-name
2.5.1.b(2)	2.6	Customer-name	3.1*	(not applic.)
2.5.1.b(3)	2.5.2	Data base set linkage	2.5.1.b (1)	(display)
2.5.1.c(2)	2.5.2	Acct-number	2.5.1	Customer-name
:	:	:	:	:

*Though not shown in the design, section 3.1 would be bill generation in batch, which would likewise update the customer's record.

Summary:

Design Inconsistencies: None

Parameter Inconsistencies: 2.5.1.c(2) : should enter 2.5.2 using Customer-name

DESIGN CONSISTENCY CHECK
Figure 11

determining which design elements are dependent upon each other. This should show which design elements a particular module is dependent upon, and which other modules are dependent upon that particular module. Methods of interface for each should also be noted, and any discrepancies pointed out. For example, a "supervisor" module should be passing the necessary parameters to a subroutine. Those parameters should be noted as interface methods, and any missing elements should be pointed out as discrepancies.

Structured walkthroughs should then be conducted to determine the correctness of the proposed design. A walkthrough is normally meant to be a peer group review of the product, and is commonly done after a program has been completed. Roles are assigned to the participants, such as presenter, coordinator, secretary/scribe, maintenance oracle, standards bearer, user representative, and reviewer, and each role carries specific responsibilities [Yourdon,1979]. The walkthrough involves a reading and visual inspection of the program logic by a team, with an actual manual execution of sample test cases. Structured walkthroughs have been proven to be effective in finding 30-70% of all logic design and coding errors [Myers, 1979].

It is to be emphasized that the type of structured walkthrough proposed here would be done by the test team. The designer may still present the design, but it will be to the test team rather than to a peer group. The chief tester will serve as coordinator of the activity.

The idea of using design and code inspections rather than walkthroughs has been proposed by Fagan [1976], claiming that error rework is more manageable and less costly with inspections than with walkthroughs. The inspection method employs a team composed of a moderator, designer, coder, and tester.

In this method, after the designer has presented an overview of the system to the entire team, the individuals study the design, logic, and intent of the system separately, using checklists of clues on finding frequent error types. The team then meets to perform the inspection, at which time a "reader" is chosen by the moderator (usually the coder). The reader is expected to paraphrase the design as intended by the designer, covering every piece of logic at least once. During this discourse, any errors noted by the team members are pointed out, classified by type, and identified as to severity (major or minor). A solution is only noted if it is obvious; the main purpose of the

inspection is to find errors, not to discover solutions. The moderator produces a final written report of the inspection and its findings.

Any errors found during inspections are reported to the proper persons for reworking, and a follow-up inspection is made of the "fixes". If more than 5% of the material has been reworked, the team should reconvene and carry out a complete inspection; otherwise, the moderator can either verify the rework himself or reconvene the team for either full or partial reinspection.

In actuality, the proposed methodology takes advantage of the better ideas of both inspections and walkthroughs. Peer review was ruled out in Chapter III as being less efficient than test team review. Parts of the design inspection, such as having each team member studying the design individually, has been kept as part of the process. When the test team meets to discuss the design, they should discuss all the errors that were detected individually (especially to determine how they might affect other parts of the design), as well as perform a manual execution of the program design logic with sample test cases.

C. Unit testing (Phase VII)

Unit testing, which appears as Phase VII of the testing methodology, is the heart of system testing. Prior to this point, the testers' main concern has been that the design documents describe the entire system as completely and as accurately as possible, and that the programmers have received a clear picture of the intended functions of the system.

During Phase VII, the testers must assure that the code does indeed perform those functions and no others. This is the most challenging and difficult task the tester will face, and requires considerable skill. Automated aids should be used whenever possible, but some understanding of the theory is necessary to use these aids well. This phase combines both static and dynamic analysis techniques. Within this section, means of performing the two, along with test data generation methods, are proposed.

1. Static analysis

Static analysis of code involves examining certain features of the code without actually executing it. A great deal of information can be obtained from certain compilers, such as variable cross-reference listings, but this

information must somehow be analyzed.

The first necessary step is to determine whether all of the appropriate design elements have been represented by the code. When the programmer hands the code to the tester, a document similar in format and purpose to the General System Design/Detailed System Design Cross Reference can be used to determine if the code is in fact consistent with the design. Each of the design elements is listed on the left-hand side; the corresponding modules are listed on the right-hand side. Any overlapping of functions, inconsistencies, or discrepancies are noted in a summary of results. Figure 12 (Code/Detailed System Design Cross Reference) illustrates use of this form.

Static analyzers (which some feel should actually be embedded in compilers [Fairley,1978]) can be used to obtain such information as syntax errors, number of occurrences of source statements by type, variable cross references, analysis of identifier usage, subroutines and functions called, uninitialized or unused variables, unexecutable code, and parameter lists. In addition, they can be used to produce a flow graph or control graph representing decision-to-decision paths within the program. Another type of graph that can be produced is a call graph (Figure 13),

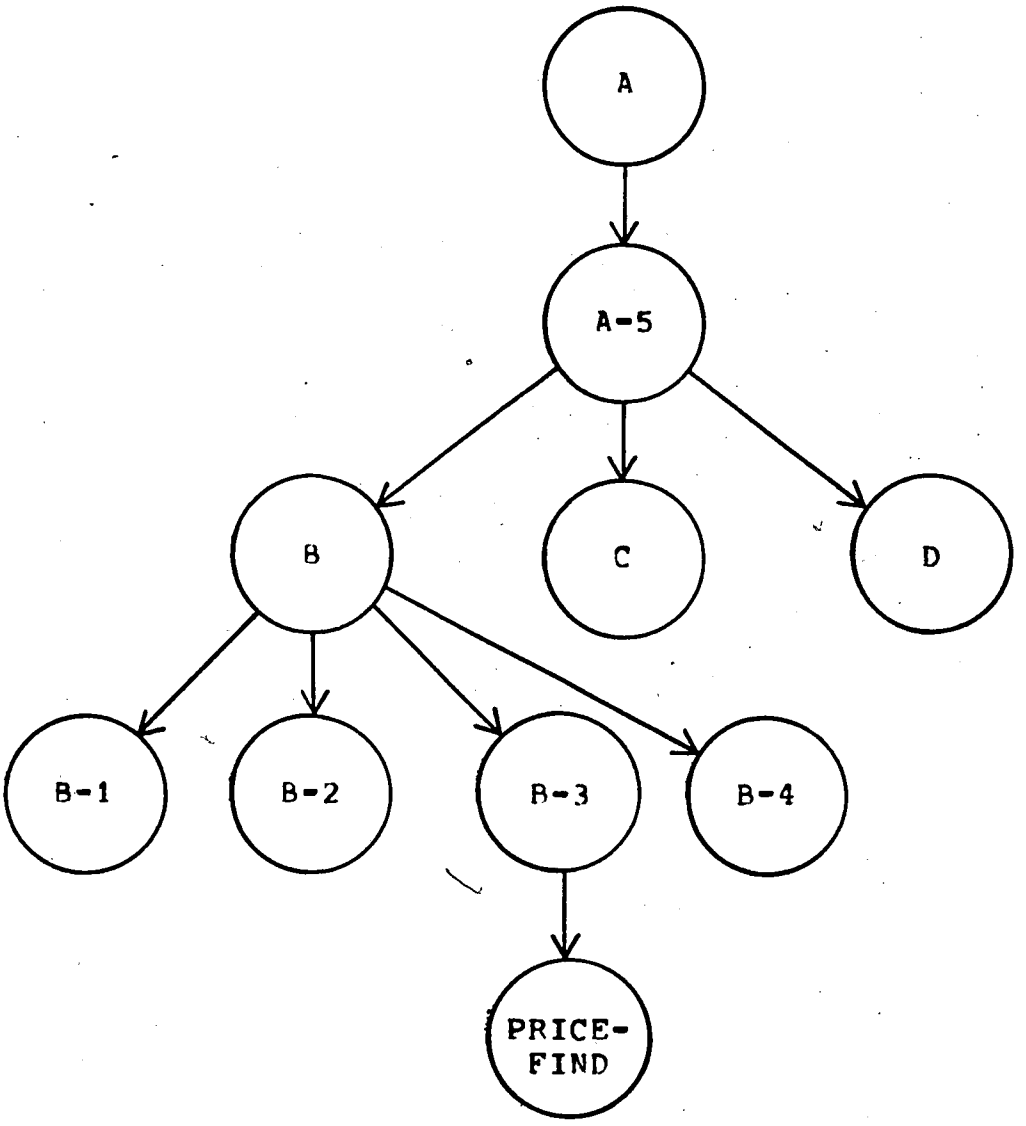
DESIGN ELEMENT	CORRESPONDING CODE ELEMENT
2.0 On-line functions	Program 1, Section A-MAIN
2.5 Item entry	Prog 1, B-Order through B-4-Old-Customer
2.5.1 Customer look-up	Prog 1, B-Order
2.5.1.a	Prog 1, B-Order, B-1-Find
2.5.1.b(1)	Prog 1, B-4-Old-Customer
2.5.1.b(2)	none
2.5.1.b(3)	Prog 1, B-4-Old-Customer
2.5.1.c(1)	Prog 1, B-2-New
2.5.1.c(2)	Prog 1, B-2-Charge
2.5.2.a	Prog 1, B-3-Item-Entry
2.5.2.b	Prog 1, B-3-Item-Entry
2.5.2.c	Prog 1, B-3-Item-Entry
2.5.2.d	Prog 1, Price-Find
:	:
:	:

Summary:

Missing Code Elements: 2.5.1.b (1) Delinquent customer check

Superfluous Code Elements: None

Figure 12

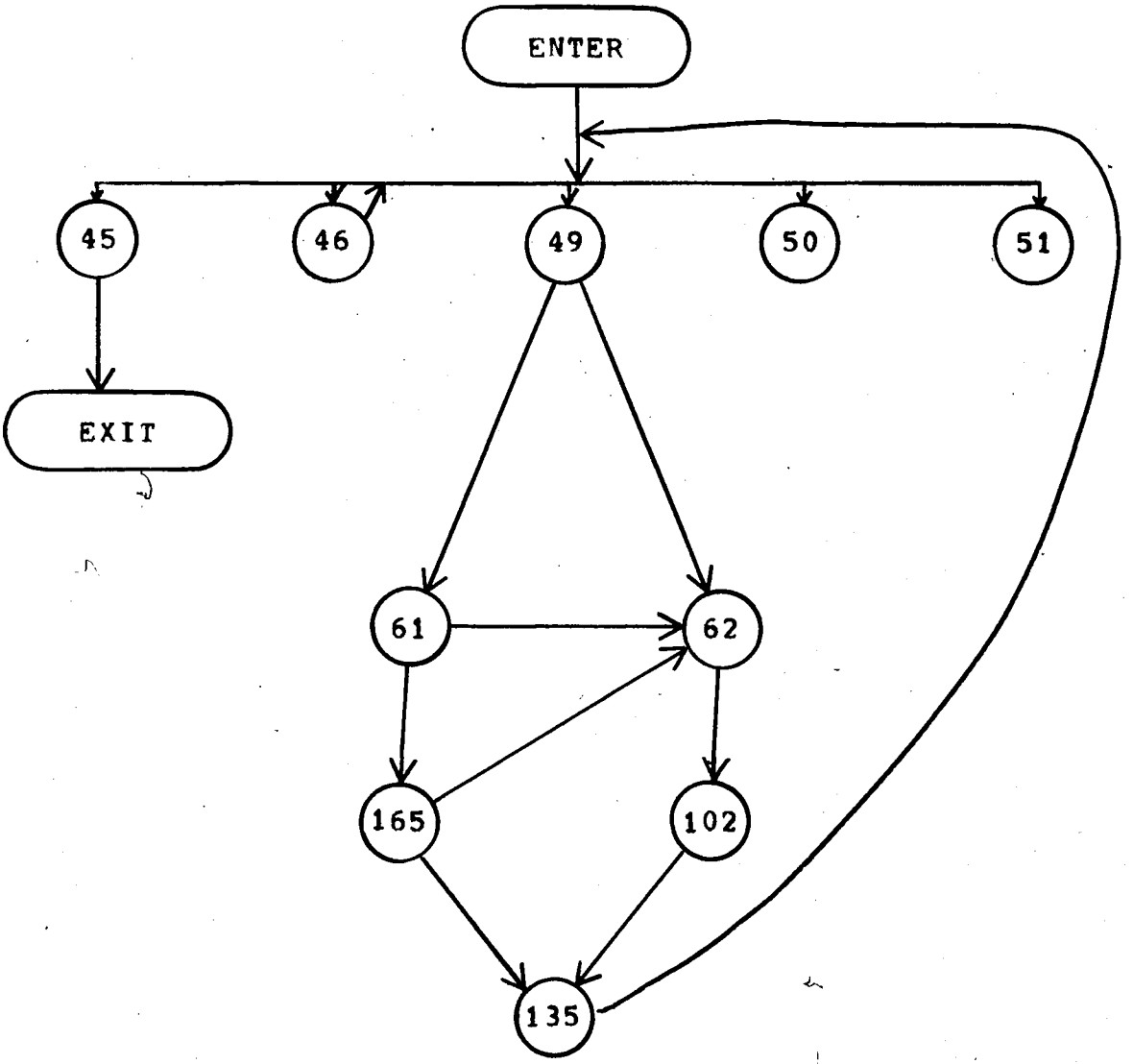


CALL GRAPH
Program 1
Figure 13

with each node representing a program unit and each arc an invocation.

Directed graphs are often used to represent the program control flow. Each node of the graph represents a decision or conditional statement, and each arc a series of statements between decisions. This is also referred to as a segment, defined as a "sequence of contiguous executable statements for which all statements will be executed if and only if the first is executed...[It] begins with the statement to which control is transferred to and ends with a statement that transfers control to an adjacent statement." [Brown,1975]. The directed graph (sometimes called digraph) is not the same as a traditional flow chart, as can be seen in Figure 14.

Osterweil [1977] proposes a method of static analysis known as data flow analysis, in which source code is searched for data flow anomalies such as uninitialized variables or initialized variables which are never referenced. The analysis is performed by creating a flow graph for each program unit and then searching each flow graph for variable patterns. It is done across subprograms, and is performed by passing over program units, analyzing each unit once. Lexical analysis is performed on each



Nodes represent program decision statements.

Numbers represent program line numbers.

DIRECTED GRAPH OF PROGRAM CONTROL FLOW
Program 1
Figure 14

source statement, and local and non-local variables are examined for anomalies. The system DAVE was developed by Osterweil to perform data flow of Fortran programs. Data flow analysis should be performed by the programmers prior to turning the code over to the testers. However, the testers should still carry out this analysis to insure that the variables are all defined, initialized, and used.

Another type of analysis for source code is discussed by Krause [1973]. Syntax analysis is performed on the code to identify statement types (assign, transfer, conditional transfer). What can be extracted from this information are segments to which each segment can transfer or will be accessible from, variables which control the branching, variables referenced within the segment, and variables computed within the segment. This becomes more meaningful when coupled with dynamic analysis techniques, as will be seen later.

Much of this work can be automated, and there are packages available which perform some type of static analysis, some of which are noted in Appendix B. Many compilers have built-in features which can be readily applied to the testing process.

After the analysis has been done, manual interpretation of the information produced may be necessary. The Variable Analysis chart (Figure 15) can be used to display the results of data flow analysis. For each variable name, it identifies its function (input, computational, output, etc.), its type (integer, real, alphanumeric, boolean), where initialized, and where used. Any anomalies, such as uninitialized or unused variables, should be noted.

The testers should have an automated static analyzer or compiler which performs data flow analysis and produces a control flow graph, a call graph, and a variable cross reference. (Other types of information, such as analysis of statements by type, are superfluous to the testing process and are more of interest to software engineering researchers.) Such tools should be considered for purchase prior to implementation of the testing methodology. If no suitable software package can be located, the test team should design a system that will perform these functions. Responsibility for building the static analyzer will either rest with the test team or with an in-house system programming group.

VARIABLE	FUNCTION	TYPE	WHERE DEFINED	WHERE INITIALIZED	LINES USED IN
ACCOUNT-NUMBER	CALC OUTPUT	N	1300	---	10700 11000 11100
CHARGE	INPUT	A/N	1200	10100	10100 10200
CUST-NAME	INPUT	N	1100	5900	5900 7200
DONE	FLAG	BOOLEAN	2000	(NA)	6400
END-OF-CUSTOMER-SET	FLAG	BOOLEAN	1800	(NA)	6000 6200
ERROR-FLAG	FLAG	BOOLEAN	1700	1700	7500 7800 13600 16900
FOUND	FLAG	BOOLEAN	1900	(NA)	6000 6100
SAME-CUSTOMER	INPUT	A/N	1400	16400	16400 16500
THE-DATE	INPUT	N	2300	---	14300
TODAYS-DATE	INPUT	N	2200	---	---
WISH	INPUT	N	1000	4400	4400 4500 4600 4900 5000 5100

Summary:

Uninitialized Variables: ACCOUNT-NUMBER, TODAYS-DATE, THE-DATE

Unused Variables: TODAYS-DATE

VARIABLE ANALYSIS
Program 1
Figure 15

2. Dynamic analysis

Dynamic analysis is a method of testing in which an attempt is made to simulate an actual environment, and then to exercise the system in a controlled and systematic way to demonstrate the presence of required functions and the absence of unwanted functions. Normally, dynamic testing is performed on individual programs or systems of programs, and begins where static analysis ends.

Dynamic analysis of programs is the monitoring of the run-time behavior of the program during execution. It requires the use of a test oracle, which is an external mechanism against which output is checked for correctness. Two types of dynamic analysis are black-box and white-box testing.

Black-box testing means that no knowledge of the program's internal structure is necessary [Howden, 1978g]. Test data can be derived from the functional specifications or from the properties of the design elements (i.e. formulae, algorithms, etc.).

White-box, or logic-driven testing [Myers, 1979] involves some knowledge of the internal program structure. It requires construction of a directed graph of the

program's control structure. Data are deliberately selected which will drive the program to exercise the various paths in the control flow structure. This is known as path testing.

A variety of path testing is branch testing, in which an attempt is made to test every conditional branch in the program. The difference between the two is that with path testing, paths are looked at only as passages from program entry to program exit, whereas in branch testing, each branch is individually considered. This could lead to excessive testing, since not all decisions lead back to similar paths; once certain data decisions have been made, some paths may become inaccessible.

An even more primitive form of testing involves testing each statement at least once. This is relatively inefficient as compared to the other two methods. Testing of all paths necessarily implies testing of all statements; there is thus no need to consider each statement separately.

An experimental study showed that path testing revealed 18 errors out of a total of 28, whereas branch testing only revealed 6 errors (statement testing was not included in the experiment). Path testing was thus concluded to be the more reliable of the two [Howden,1978c]. Path testing was

therefore selected for inclusion in the proposed methodology. Figure 16 depicts a typical path testing plan for the program flow shown in Figure 14.

Probes can be inserted either as additional statements or as calls to statistics-gathering subroutines. This technique requires a preprocessor for inserting the probes and a post-processor for collecting statistics during program execution. The execution can be monitored, and terminated if control has not been transferred in a prescribed direction. Such information as ranges of variables, variables undefined but referenced, variables defined but never referenced, and changes in variable values can be recorded using statement probes. With regard to control flow, it is possible to keep track of path, branch, and statement traversals to determine what percent of the program is actually executing, and with what frequency [Huang,1978].

Probes can be most effectively used when consideration is given to optimizing their placement and installing a minimal number of general monitors at suitable locations. Ramamoorthy [1975] proposes a method for finding the minimal set of arcs that result in a directed path, and placing a monitor at each node along the path. Here, the directed

Some Possible Paths:

1) 52 - 64 - 168 - 138 - 48
2) 52 - 65 - 105 - 138 - 48
3) 52 - 64 - 168 - 65 - 138 - 48
4) 49 - 48

Decision Points	Values needed to follow path
48	WISH = "E"
49	WISH < 1 or > 3
52	WISH = 1
64	ERROR-FLAG = 2(FOUND)
65	ERROR-FLAG = 1(END-OF-CUSTOMER-SET)
105	CHANGE = "Y"
138	ITEM-TYPE = "END"
168	SAME-CUSTOMER = "N"

Paths	Data to exercise paths
1	1-2(FOUND)-N-END-E
2	1-1(END-OF-CUSTOMER-SET)-N-END-E
3	1-2-N-1-N-END-E
4	4-E

PATH TESTING PLAN
Program 1
Figure 16

graph method is a necessary precondition for carrying on this technique. In the proposed methodology, probes should be used to keep track of path traversals. The information provided will be useful in deciding when a program has been tested sufficiently, according to the acceptance criteria of the Unit Test Plan.

Several automated tools have been developed which perform some type of dynamic analysis (see Appendix B). Noteworthy in all these tools is their dependence on the source language, normally Fortran. Nothing appears to be available which performs any type of analysis on COBOL, PL/1, PASCAL, or other common programming languages.

Tool development requires implementation of the theory. The ideal tool will contain a static analyzer to perform data flow analysis and construct a directed graph; a test data generator which analyzes that graph and produces an optimal set of test cases; and a dynamic analyzer which monitors run-time behavior and produces post-execution statistics. It should of course be able to perform this analysis on the source language. Since most of the systems available work only on Fortran, a company which uses another language predominantly should consider its own development of such an automated tool.

In the proposed methodology, both black-box and white-box methods are recommended to insure more thorough testing. The test data generation methods discussed in the next section insure that both black-box and white-box testing are performed.

3. Test data generation

One of the most important testing activities is selecting data which will thoroughly test the system. The ability to carefully and critically select test cases which exercise all possible paths while keeping the number of test cases manageable is a difficult (and some say impossible) task. This is where a trained tester should develop his strongest skills.

Test cases can be derived both from functional specifications and from structural analysis of programs. The former is an example of black-box testing, where the tester is only concerned with the functioning of the system, without regard for its internal structure. Test case design from structural analysis provides for white-box testing. Thus, the total cases derived from both analyses should be combined to produce the data set. Test data can be live or contrived, and can either be selected randomly or according to some predetermined methodology.

a. Randomly selected data

When data are selected randomly, the selection is done within some range of values or within some particular type of data structure. Several theorists feel that random test cases do not give statistically significant results [Huang,1977]. Acceptance sampling and other forms of traditional random sampling techniques cannot be successfully applied to programs and systems, where more errors are likely to occur with values at or beyond the extremes of data ranges than with normal data. Programs which appear to function normally may be doing so only for valid data. It is therefore crucial to test the program for invalid or abnormal cases, in which case randomly selected data will not necessarily exercise all the possible conditions. Thus, the use of random test data selection is ruled out as a desirable option.

b. Equivalence class partitioning

Myers [1979] proposes a method of data selection known as "equivalence partitioning", whereby the total range of possible input domain is partitioned into classes of equivalent data. Each class consists of a range of values all of which are valid, along with those which are invalid. For example, a variable valid for the range 0-100 (such as a

percentage), would give rise to one valid class and two invalid classes (one for values < 0 and one for values > 100). One test case should then be generated for each valid and each invalid input class. In this example, three test cases would include:

(1) A value between 0 and 100, say 45;

(2) A value < 0, say -5; and

(3) A value > 100, say 105.

In this way, normal values are not tested excessively, whereas extraordinary values are tested thoroughly. In addition to having a test case for each equivalence class, there should also be one for the ends of all ranges -- i.e., minimum and maximum values. The two additional cases required in our previous example would be 0 and 100. This refinement is known as "boundary value analysis" [Myers,1979].

This method would work well for deriving functional test data sets for black-box testing. The testers should therefore use this method prior to unit testing. These functional data sets will be dependent only on the Detailed System Design documents, so that no knowledge of the

internal program structure is necessary.

c. Determination of optimal set of test cases for control flow exercise

The test cases which exercise program control flow are derived from the predicates along a path from entry to exit. This set of path predicates will be traversed if all the branch predicates are satisfied at least once. Each branch predicate is considered individually to determine a set of data values necessary to follow the given path. Goodenough and Gerhart recommend usage of a condition table, listing each possible combination of conditions that can occur [Goodenough,1975]. Each combination is called a test predicate, and the claim is made that a program is completely tested when a data case has been selected to satisfy each test predicate. The test predicates must be mutually independent, and together must represent every branch and every potential termination condition.

Unfortunately, the number of test cases can become unmanageable when an attempt is made to test all possible combinations of conditions, especially if the program is large and contains several branches. Exhaustive testing is impossible, so rather than attempt to develop an exhaustive set of test cases, it is more practical to select an optimal

number of cases which will exercise the software thoroughly enough to detect most of the errors. The overall number of test cases can be substantially reduced using control path analysis, since certain combinations of inputs will be impossible and some paths can become unexecutable once certain data have been selected. As testing proceeds, the number of errors detected will ideally decline to the point where it is no longer cost effective to continue testing.

d. Use of symbolic evaluation

A recent breakthrough in the theory of test case selection is the notion of symbolic execution, which introduces symbols as input values to represent some fixed but unknown value. With symbolic evaluation, all arithmetic computations are delayed or generalized by derivation of an algebraic formula. When a condition or decision statement is reached, the properties which must be satisfied in order to follow a certain path are conjoined to form what is called a "path condition". The inputs must thus satisfy all the properties of the path condition to follow its associated path. The final value of the variable is an algebraic combination of the various manipulations performed on the variable, rather than a real or integer value. Path conditions will be either true or false, depending on the

value of the variable. Each symbolic execution result is equivalent to a large number of test cases, thus eliminating the need to test for all unique inputs over a class of data. [King,1976]

The result of symbolic execution is a series of inequality and equality constraints on the input variables. These can be analyzed using linear programming, since the constraints define subsets of the input spaces which will execute each path. Ramamoorthy [1976] describes a method for generating test data using this procedure.

Symbolic evaluation techniques are recommended for inclusion in the proposed methodology. Each of the predicates necessary to exercise a path can be derived using the control graph generated during static analysis. The resulting expressions will then be used to compile test cases which are combined with the functionally derived test cases to produce an optimal set.

e. Maintaining test case data for retesting

It is rarely the case that a program executes without errors the first time. Since test cases have been so carefully derived and results recorded, some thought should be given to maintaining the same test cases to use while

modifications are made to the modules. The test cases should be stored in a data base or data file, along with the results of execution and the program version which produced those results. This data base will then become a historical record of the system and its modifications, and results of various modifications can be compared to determine if the change had any effect on the system's operation.

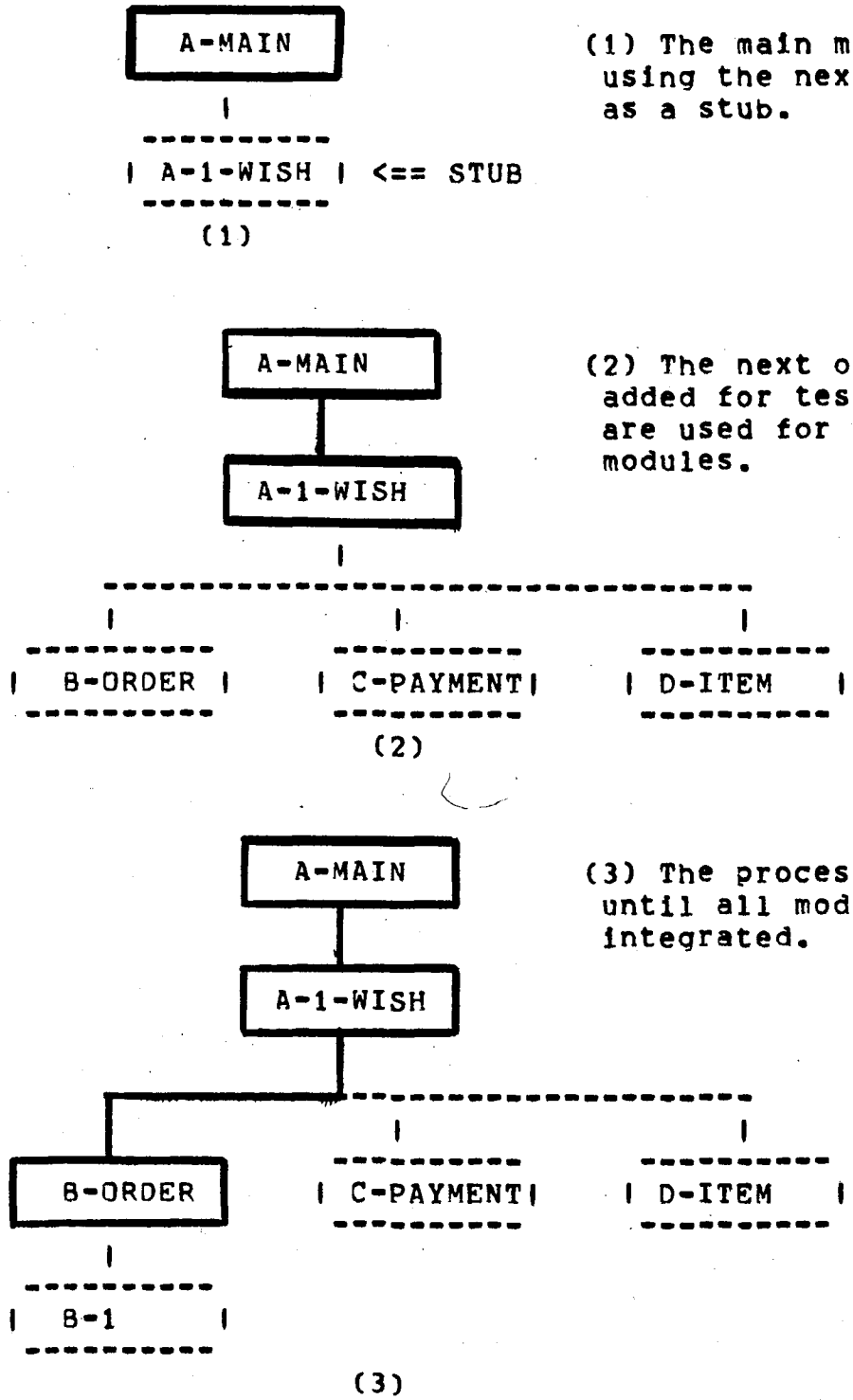
The best recourse is to either purchase or develop a testing system which performs as many of the recommended functions as possible, including: directed graph production; symbolic evaluation; path testing; probe insertion for statistics gathering; and maintenance of test cases and program execution history in a data base. By automating this process, a great deal of time can be saved. In addition, the procedures will be standardized and the results more reliable.

D. Integration testing (Phase VIII)

When the modules have been thoroughly tested, integration of the units can proceed. This comprises Phase VIII of the proposed methodology. The units can be combined in a top-down or a bottom-up fashion.

With top-down testing, the control module is tested first and stubs are used for lower order modules (Figure 17). When the main module has been sufficiently tested, next-order modules are added one at a time, and testing proceeds until all the modules have been added.

Bottom-up testing starts with the lower order modules (Figure 18). In this case, driver modules with test inputs which call the individual module are needed. Next higher modules are then added, until the entire system/subsystem has been constructed.

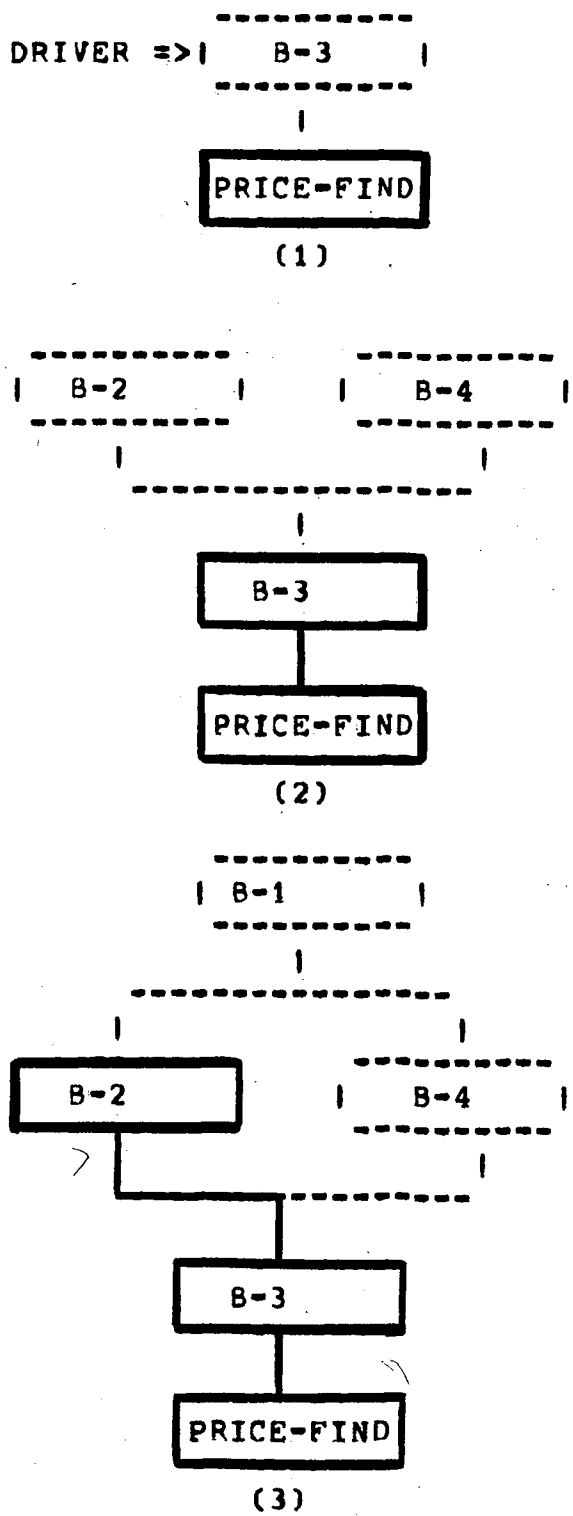


(1) The main module is tested, using the next order module as a stub.

(2) The next order module is added for testing, and stubs are used for lower order modules.

(3) The process continues, until all modules have been integrated.

TOP-DOWN TESTING
Figure 17



(1)The lowest level module is tested first, using the next higher level as a driver.

(2) The next higher level is added, using higher level modules as drivers.

(3) The process continues, until the highest level module has been integrated.

BOTTOM-UP TESTING
Figure 18

The following charts summarize some of the advantages/disadvantages of each method [Myers, 1979]:

Advantages

Top-down	Bottom-up
1. Best if major flaws exist in main logic.	1. Best if major flaws exist in lower modules.
2. Once the major I/O modules have been added, representation of test cases is easier.	2. Test conditions are easier to create (no need for I/O stubs).
3. Skeletal program (system) exists early on.	3. Observation of results is easier.
4. Minimizes the system integration problem.	

Disadvantages

Top-down	Bottom-up
1. Need to create stubs.	1. Need to create drivers.
2. Representation of test cases in stubs can be difficult to create.	2. Program (system) does not exist as entity until late in testing stage.
3. Test conditions are difficult to create.	
4. Observation of output is difficult.	
5. Deferred completion of testing certain modules (discourages thorough testing of suprogram modules).	
6. High cost due to repetitive testing.	

Either top-down or bottom-up testing can be tried when implementing the proposed methodology, since the advantages and disadvantages are relatively equal, and neither method stands out as "best". However, one of the two should be selected and carried through on a single project.

During integration testing, a static analyzer should be used to generate a call graph. This is necessary to determine flow among the various units, and should be compared to the original integration test plan to determine if the module interface matches the original design. The highest level module will be that which is not called from any other module, whereas the lowest level modules do not call any other modules. The Integration Test Plan should be altered (if necessary) to account for any discrepancies before integration testing can begin.

Once the call graph has been generated, the methods of interface should be compared. If the parameters or file structures are common to two modules, they should be of the same type, name, or any other mode necessary to their proper interface.

To perform the actual testing, path execution is again recommended. However, it is not necessary to retest the logic of the individual units; this would be cumbersome and time-consuming. The interface parameters should be used to generate test cases, and only paths which modify those parameters need be executed. Structural analysis and symbolic execution techniques can be used to locate those paths and derive test cases. If two units interface with a single parameter, the number of test cases needed to exercise that interface may be very small.

The test cases should be stored in a data base, as with unit testing. As a new unit is added, it is necessary to test only that new interface, and therefore necessary to only use the set of test cases which exercise all possible means of reaching the new module from its calling unit. Testing every possible path to reach that point would be superfluous, since every path should have already been tested.

Proceding in this manner, errors will be localized to those modules being added, specifically to the interface. Those errors should be corrected before continuing. When the final units are added, the entire data base of test cases (or, if it is impossible to run the entire set, a

subset of the data base) should be rerun as added assurance that the system functions correctly.

E. System performance testing (Phase X)

When the developer has reached Phase X, he should be relatively satisfied that the system is performing accurately and as the client desired. The system is practically ready for installation, and no major problems should remain. If the hardware which will house the system differs from that used to develop the system, the switchover should be made at this point, placing the system in an operational situation.

1. Usability testing with regard to human factors

A system is for its users, and serves no purpose unless it can be learned and operated by its users. Perhaps the best way to determine a system's usability is to let the ultimate users (non-systems personnel) work with the system. They should be provided with some training and materials (i.e. a user's manual). During this test, a representative of the test team should be on hand to note the users' reactions and to aid in the event of unforseen consequences; however, the user should be left alone for the most part to figure out the system. A real environment should be

simulated as closely as possible, and the testing should go on for at least a week to note the user's reaction as he/she "learns" the system. While observing the user, some points which should be noted are:

(a) Does the user understand the prompts?

(b) Are the reports or responses generated by the system comprehensible to the user?

(c) If the user enters erroneous data, does the system respond with a clear enough message so that the user understands the error?

(d) Does the user seem to enjoy working with the system?

(e) Is the manual referred to more than is necessary?

(f) Does the use of the manual decrease with time?

(g) Does the user actually understand better after having consulted the manual? etc.

Many of these questions can be answered from mere observation. A user questionnaire could be a useful tool in determining system usability.

Familiarity with the system is that point where the user can perform common transactions quickly, without having to refer to a manual. The tester should note both the ease with which the system is "learned" and the ease and comfort with which it is operated once it has been learned.

2. Security testing

Security testing, used here to mean procedural rather than physical security, is especially important where there is a cash flow or where different personnel levels have access to various parts of a common data base. All levels of passwords, security checks, etc. should be tested carefully to insure that those persons can perform only tasks meant for them, and no more. Methods of file update should have been cleared with the client in the early design stages so that no unauthorized persons can access, update, or delete the files. At this point, security checks for those functions should have been built into the system, and all that remains is insuring their accurate performance. Figure 19 shows a report which can be used with security checks.

Once the users have become familiar with the system (after the first week of system testing), they can be "dared" to be devious and to attempt to perform unassigned

LEVEL*	FCTNS ALLOWED**	FCTNS PERFORMED	DEVIATIONS
1	all	all	OK
2	1-4,6,7	1-4,6,7	OK
3	1-4	1-4,7	1 Extra Fctn
4	1-3	1,2	1 Missing Fctn
5	2	1,2	1 Extra Fctn

*Levels as defined by organization
1 = Owner
2 = Store Manager
3 = Store Clerk
4 = Central Facility Supervisor
5 = Central Facility Clerk

**Functions:
1. View customer report
2. View item report
3. Place an order
4. Accept payment
5. Update price tables
6. Override payment
7. Override delinquent customer message

Summary of discrepancies:

Level 3 can perform Function 7

Level 4 cannot perform Function 3

Level 5 can perform Function 1

SECURITY CHECK REPORT
Figure 19

functions. The user may have more foresight than the developer in breaking security; if it is presented as something of a game, future machinations may be avoided.

The tester should also examine interdepartmental relationships within the organization during these phases to determine potential areas of security slips. It might be discovered, for example, that two different departments accessing different areas of the data base are closely related and prone to exchanging password information. Extra checkpoints should be installed, e.g., by issuing badges, if the security system is not foolproof.

3. Operation of the system under stress and heavy volumes of data

For this, the system should be subjected to a heavier than normal load to determine its maximum capacity and to study both system performance and user response during peak periods. If it is intended to be accessible to ten users, for example, tests should be run with ten operators all trying to access and/or update the system simultaneously. The best time to do so is one or two weeks after the user has become familiar with the system and appears to be relatively comfortable with its normal operation. Extra terminals should be added temporarily where possible to

analyze the system's potential for growth. The amount of data entered should be greater than usual, and extreme cases should be considered. Any system failures or signs of stress should be carefully analyzed and noted in a report.

4. Hardware performance testing

The system's performance should be examined while some of the aforementioned tests are taking place. Response time is important, and the user's reaction to a lengthy response time can become increasingly important to note during stress testing. Another performance factor which can be examined is throughput rates using normal workloads.

If the hardware configuration has been predetermined, the system's operational performance can only be judged against that particular configuration. If the configuration is fairly flexible, various combinations should be tried to optimize the usage of both the computer and the peripherals. The portability of the system should also be analyzed at this point.

5. Recovery testing

The only way to perform recovery testing is to force a system crash and create the worst possible conditions while doing so (i.e. the data base is in a transitional phase).

Recovery techniques, which should have been designed into the system, can be tested to see how effectively they work. The status of the data files both prior to and after the crash should be carefully compared, particularly those files which were being modified.

This concludes the discussion of the proposed methodology, along with suggestions for its implementation. A summary chart of those recommended tools/techniques along with phases of implementation follows this chapter. Chapter V, in summation, discusses what has been proposed in terms of its benefits to the software and system development industry.

Phase	Technique/Tool
II. General System Design Analysis	Customer Requirements/General System Design Cross Reference General System Design Evaluation
IV. Detailed System Design Analysis	General System Design/Detailed System Design Cross Reference Detailed System Design Evaluation Design Consistency Check Structured Walkthrough
VII. Unit test	Static analysis: Code/Detailed System Design Cross Reference Data flow analysis Variable Cross Reference Call graph Control flow graph Dynamic analysis: Functionally derived test data Symbolic evaluation Path testing Probe insertion Data base of execution history
VIII. Integration test	Top-down integration
X. Performance Test	Usability test Security test Stress test Recovery test Hardware performance test

RECOMMENDED TOOLS AND TECHNIQUES
Figure 20

Chapter V. Conclusions

If one were to ask which phase of a system's life cycle consumed the most in both dollars and time, yet yielded the least in terms of benefits, the answer would most likely be maintenance. The proportion of time and money claimed by system maintenance is exorbitantly high.

The most apparent reason for this inordinate expenditure is the lack of coordinated effort during the design and development stages. Had the design been well examined and the system well tested prior to its installation, many later problems could have been eliminated, or at least mitigated. In this light, even the slightest bit of attention paid towards quality assurance could improve a system's performance markedly, and could eventually raise the level of confidence and respect paid to the industry by its clientele.

Most people in this country are affected in some way by computer-based information systems, and many major decisions are made based on such systems. The number of "man-rated" systems (i.e. those in which a system failure could result in the loss of a human life) is steadily increasing, and the quality of such systems must be excellent. A payroll, banking, or any other type of financial system can be

responsible for severe financial losses if it is not carefully tested prior to use by the client. It is critical that the reliability and accuracy of such systems be subjected to severe scrutiny before they are put into operation. The client should not be the one to discover the "bugs".

It seems obvious that testing a system well would be to the advantage of the software developer, and yet it is not being done. Three reasons have been identified as to why this is so:

1. Programmers (or systems people) don't like to do testing.

This can be termed a "people" problem. Even if a programmer were supplied with all of the tools and know-how necessary to perform testing efficiently, he/she would probably still not enjoy the task. A similar problem already exists with documentation, even when the procedures and methods are very clearly laid out. The proposed methodology attempts to solve this problem by taking testing out of the programmer's hands, making it a completely separate profession. This may not solve the attitude problem, but it would at least assure that testing is being performed. Perhaps with better tools and procedures, attitudes will eventually improve.

2. System development is usually done on a crisis basis, rather than being planned.

Perhaps the industry is trying to grow too rapidly, and is suffering from severe growing pains. There is no need to develop a system as quickly as possible. If the operation has been functioning for years in a manual or semi-automated fashion, it can continue to do so for a few extra months. A client is better off waiting a little longer for a better product than trying to work around a system with problems. Getting a system to perform its intended functions is much easier than assuring that the system does not perform any unintended functions; it's the unintended functions that can create the most problems. Within this methodology, the idea of planning for system development and for testing, of extending the time for development if necessary, and of providing for careful checks along the way has been proposed.

3. Most developers do not know how to test thoroughly.

Testing may be generally misunderstood, but a great deal of progress has been made in the evolution of testing theory, and in its implementation. This approach to testing sets forth a step-by-step means for bringing the theory into

the field. It is certainly not the ultimate answer, but it is at least a beginning. Though system developers may not know how to test, or even when or what to test, the approach developed in this paper provides a rational and a systematic method for attacking the problem.

The preceding pages have described a methodology which makes testing an integral part of the software development process. Rather than testing at only one point, it prescribes a series of checks throughout development to create a more controlled procedure and to assure more accurately designed and implemented systems.

Another major feature of the methodology is the creation of a chief tester team, with clearly defined responsibilities. Members of this team will become experts in testing theory and techniques, and will develop automated tools based on that theory to implement the methodology.

This synthesis of the concepts of test theory and system development procedures has produced an integrated system development/system testing methodology which can be used in any type of software development organization and which will aid in the production of better quality systems.

This first step in software quality assurance could help change the industry in a number of ways. In addition to increased product reliability, it might lead to the development of a professional specialty, and even to the development of a specialized section of the industry. The formalized software audit service [Miller,1976], separate from the software development organization, would centralize software quality management by setting standards and "certifying" software as acceptable according to those standards. Once a software product has been certified, the service bureau would maintain a copy of the software, and require modifications to that system to be resubmitted for re-certification. One advantage to a development of this sort would be the improvement of software quality resulting from trained professionals' controlling the testing and approval processes.

Since there would be a cost associated with such a service, organizations would tend to be more careful in the design and development process. They would certainly rather their software be certified as quickly and cheaply as possible. As the number of certified systems would grow, software customers would begin to insist on certified software. This would increase the number of developers using such a service.

Although the notion of a software audit service could be a natural outgrowth of the internal testing team approach, or of employing outside consultants to perform these same services, it is yet to be realized. Currently, the problem must be solved by software and systems developers. The software audit service is a viable concept, but we cannot afford to wait for it to become a reality; developers must begin formulating their own standards for quality and must begin to establish procedures to assure that those standards are met.

This paper has suggested one means of doing so. In terms of benefits, implementing this methodology might mean dollar savings in the long run, since less time should be devoted to maintenance. However, this can only be assumed here, as it has not been proven. A future area of research might be the application of this methodology in a controlled environment to determine its cost-effectiveness. Empirical studies of any of the suggested techniques or procedures to refine the methodology would also be a useful contribution.

BIBLIOGRAPHY

[Alberts, 1976] Alberts, David S. The economics of software quality assurance. National Computer Conf, AFIPS Conf Proc 45: 433-42, 1976.

[Baker, 1972] Baker, F. T. Chief programmer team management of production programming. IBM Systems Journal 11(1): 56-73, 1972.

[Bate, 1978] Bate, Roger R.; Ligler, George T. An approach to software testing: methodology and tools. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 476-80.

[Boehm, 1975] Boehm, Barry W. The high cost of software. Practical Strategies for Developing Large Software Systems (E. Horowitz, ed.) Reading, Mass.: Addison-Wesley, 1975, p. 3-14.

[Boehm, 1975a] Boehm, Barry W.; McClean, Robert K.; Urfrig, D. B. Some experience with automated aids to the design of large-scale reliable software. IEEE Trans on Software Engineering SE-1(1) : 125-33, 1975.

[Boehm, 1976] Boehm, Barry W.; Brown, J.R.; Lipow, M. Quantitative evaluation of software quality. Intl Conf on Software Engineering, 2nd, Long Beach, CA: IEEE Computer Soc, 1976, p. 592-605.

[Boyer, 1975] Boyer, Robert S.; Elspas, Bernard; Levitt, K. N. SELECT -- A formal system for testing and debugging programs by symbolic execution. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 234-45.

[Brown, 1975] Brown, J. R.; Lipow, M. Testing for software reliability. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 518-27.

[Buckley, 1973] Buckley, Fletcher J. Software testing: a report from the field. Symp on Computer Software Reliability, N.Y.: IEEE, 1973, p. 102-6.

[Budd, 1978] Budd, Timothy A. et al. The design of a prototype mutation system for program testing. National Computer Conf, AFIPS Conf Proc 47: 623-27, 1978.

[Budd, 1979] Budd, Tim; Majoras, M.; Sneed, H.; Experiences with a software test factory. COMPCON '79, N.Y.: IEEE, 1979, p. 319-29.

[Carey, 1977] Carey, Robert; Bendick, Marc. The control of a software test process. Intl Computer Software and Applications Conf, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 327-33.

[Cheatham, 1978] Cheatham, T. E.; Townley, Judy A. Program analysis techniques for software reliability. Workshop on Reliable Software (P. Raulefs, ed.). Munchen-Wein: Hanser, 1978, p. 9-17.

[Chen, 1978] Chen, Wen-Tsuen; Ho, Jone-Ping; Wen, Chia-Hsien. Dynamic validation of programs using assertion checking facilities. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 533-8.

[Clarke, 1976] Clarke, L. A. A system to generate test data and symbolically execute programs. IEEE Trans on Software Engineering SE-2(3): 215-22, 1976.

[Clarke, 1978] Clarke, Lori A. Testing: Achievements and frustrations. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 310-14.

[Darringer, 1978] Darringer, John A.; King, James C. Applications of symbolic execution to program testing. Computer 11(4): 51-60, 1978.

[DeMillo, 1978] DeMillo, Richard A.; Lipton, Richard J.; Sayward, Frederick G. Hints on test data selection: help for the practicing programmer. Computer 11(4): 34-41, 1978.

[Endres, 1978] Endres, A.; Glatthaar, W. A complementary approach to program analysis and testing. European Cooperation in Informatics. Information Systems Methodology. (Lecture Notes in Computer Science, v. 65). Berlin: Springer-Verlag, 1978, p. 380-401.

[Fagan,1976] Fagan, M. E. Design and code inspection to reduce errors in program development. IBM System Journal 15(3): 182-211, 1976.

[Fairley, 1975] Fairley, Richard E. An experimental program testing facility. IEEE Trans on Software Engineering SE-1(4): 350-57, 1975.

[Fairley, 1978] Fairley, Richard E. Tutorial: static analysis and dynamic testing of computer software. Computer 11(4): 14-23, 1978.

[Fife, 1977] Fife, Dennis W. Computer Software Management: a primer for project management and quality control (NBS-SP 500-11). Wash, D.C.: National Bureau of Standards, 1977.

[Fischer, 1977] Fischer, Kurt F. A test case selection methodology for the validation of software maintenance modifications. Intl Computer Software and Applications Conf, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 421-26.

[Fischer, 1979] Fischer, Kurt F.; Walker, Michael G. Improved software reliability through requirements verification. IEEE Trans on Reliability R-28(3): 233-40, 1979.

[Fosdick, 1975] Fosdick, L. D.; Osterweil, L. J. DAVE--A Fortran program analysis system. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 329-35.

[Fujii, 1977] Fujii, Marilyn S. Independent verification of highly reliable programs. Intl Computer Software and Applications Conf, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 38-44.

[Gannon, 1978] Gannon, Carolyn. JAVS: A Jovial automated verification system. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 539-44.

[Gerhart, 1976] Gerhart, Susan; Yelowitz, Lawrence. Observations of fallibility in applications of modern programming methodologies. IEEE Trans on Software Engineering SE-2(3): 195-207, 1976.

[Gmeiner, 1978] Gmeiner, L. Dynamic analysis and test data generation in an automatic test system. Workshop on Reliable Software (P. Raulefs, ed.). Munchen-Wein: Hanser, 1978, p. 31-48.

[Good, 1975] Good, Donald I.; London, Ralph L.; Bledsoe, W. W. An interactive program verification system. IEEE Trans on Software Engineering SE-1(1): 59-67, 1975.

[Goodenough, 1975] Goodenough, John B.; Gerhart, Susan L. Toward a theory of test data selection. IEEE Trans on Software Engineering SE-1(2): 156-73, 1975.

[Hallin, 1978] Hallin, T. G.; Hansen, R. C. Toward a better method of software testing. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 153-57.

[Hamlet, 1977] Hamlet, Richard G. Testing programs with the aid of a compiler. IEEE Trans on Software Engineering SE-3(4): 279-90, 1977.

[Hamlet, 1978] Hamlet, Richard. Test reliability and software maintenance. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78) N.Y.: IEEE, 1978, p. 315-20.

[Hartwick, 1977] Hartwick, R. Dean. Test planning. National Computer Conf, AFIPS Conf Proc 46: 285-94, 1977.

[Henderson, 1977] Henderson, Peter. Structured program testing. Current Trends in Programming Methodology (R.T. Yeh, ed.). Englewood Cliffs, N.J.: Prentice-Hall, 1977, p. 1-15.

[Hennell, 1978] Hennell, M. A.; Woodward, M. R.; Hedley, D. Towards more advanced testing techniques. Workshop on Reliable Software (P. Raulefs, ed.). Munchen-Wein: Hanser, 1978, p. 19-30.

[Hetzel, 1973] Hetzel, W. C. Program test methods. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

[Heuermann, 1974] Heuermann, C. A.; Myers, G. J.; Winterton, J. H. Automated test and verification. IBM Tech Disclosure Bull 17(7): 2030-35, 1974.

[Hice, 1974] Hice, G. F.; Turner, W. S.; Cashwell, L. F. System Development Methodology. Amsterdam: North-Holland Pub Co, 1974.

[Hodges, 1976] Hodges, B. C.; Ryan, J. P. A system for automatic software evaluation. Intl Conf on Software Engineering, 2nd, Long Beach, CA: IEEE Computer Society, 1976, p. 617-23.

[Hoffman, 1975] Hoffman, Robert H. NASA/Johnson space center approach to automated test data generation. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 336-41.

[Howden, 1975] Howden, W. E. Methodology for the generation of program test data. IEEE Trans on Computers C-24(5): 554-60, 1975.

[Howden, 1975a] Howden, W. E.; Laub, J. Automatic case analysis of programs. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 347-52.

[Howden, 1976] Howden, W. E. Experiments with a symbolic evaluation system. National Computer Conf, AFIPS Conf Proc 45: 899-905, 1976.

[Howden, 1976a] Howden, W. E. Reliability of path analysis testing strategy. IEEE Trans on Software Engineering SE-2(3): 208-15, 1976.

[Howden, 1977] Howden, William E. Reliability of symbolic evaluation. Intl Computer Software and Applications Conf, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 442-47.

[Howden, 1977a] Howden, W. E. Symbolic testing and the DISSECT symbolic evaluation system. IEEE Trans on Software Engineering SE-3(4): 266-78, 1977.

[Howden, 1978] Howden, William E. A survey of static analysis methods. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 82-96.

[Howden, 1978a] Howden, W. E. DISSECT -- A symbolic execution and program testing system. IEEE Trans on Software Engineering SE-4(1): 70-73, 1978.

[Howden, 1978b] Howden, William E. Empirical studies of software validation. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 280-85.

[Howden, 1978c] Howden, W. E. An evaluation of the effectiveness of symbolic testing. Software--Practice and Experience 8(4): 381-97, 1978.

[Howden, 1978d] Howden, W. E. Functional program testing. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 321-25.

[Howden, 1978e] Howden, W. E. Introduction to the theory of testing. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 16-19.

[Howden, 1978f] Howden, W. E.; Eichhorst, H. P. Proving properties of programs from program traces. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 46-56.

[Howden, 1978g] Howden, William E. A survey of dynamic analysis methods. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 184-206.

[Howden, 1978h] Howden, W. E. Theoretical and empirical studies of program testing. IEEE Trans on Software Engineering SE-4(4): 293-98, 1978.

[Huang, 1975] Huang, J. C. An approach to program testing. Computing Surveys 7(3): 113-28, 1975.

[Huang, 1977] Huang, J. C. Error detection through program testing. Current Trends in Programming Methodology (R.T. Yeh, ed.). Englewood Cliffs, N.J.: Prentice-Hall, 1977. Vol II, p. 16-43.

[Huang, 1978] Huang, J. C. Program instrumentation and software testing. Computer 11(4): 25-32, 1978.

[Itoh, 1973] Itoh, Daiju; Izutani, Takao. FADEBUG-I, a new tool for program debugging. Symp on Computer Software Reliability, N.Y.: IEEE, 1973, p. 38-43.

[Jessop, 1976] Jessop, W. H. et al. ATLAS: An automated software testing system. Intl Conf on Software Engineering, 2nd, Long Beach, CA: IEEE Computer Society, 1976, p. 629-35.

[King, 1975] King, James C. A new approach to program testing. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 228-33.

[King, 1976] King, J. C. Symbolic execution and program testing. Communications of the ACM 19(7): 385-94, 1976.

[Kopetz, 1975] Kopetz, H. On the connections between range of variable and control structure testing. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 511-17.

[Krause, 1973] Krause, K. W.; Smith, R. W.; Goodwin, M. A. Optimal software test planning through automated network analysis. Symp on Computer Software Reliability, N.Y.: IEEE, 1973, p. 18-22.

[Krause, 1978] Krause, K. W.; Diamant, L. W. A management methodology for testing software requirements. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 749-54.

[Lyons, 1977] Lyons, N. R. An automatic data generation system for data base simulation and testing. Data Base 8(4): 10-13, 1977.

[Miller, 1974] Miller, E. F. et al. Structurally based automatic program testing. Electronic and Aerospace Systems Conf (EASCON '74), N.Y.: IEEE, 1974, p. 134-39.

[Miller, 1975] Miller, E. F.; Melton, R. A. Automated generation of testcase datasets. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 51-58.

[Miller, 1975a] Miller, E. F. RXVP: An automated verification system for Fortran. Computer Science and Statistics: Symp on the Interface, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 328.

[Miller, 1976] Miller, Edward F. A service concept for software auditing. NSF Software Auditing Workshop, 1976.

[Miller, 1977] Miller, Edward F. Notes on management and control of testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 221.

[Miller, 1977a] Miller, Edward. Notes on planning and measurement in testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 201-202.

[Miller, 1977b] Miller, Edward F. Notes on research and development of testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 233.

[Miller, 1977c] Miller, Edward F. Notes on the philosophy of testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 1-3.

[Miller, 1977d] Miller, Edward F. Notes on the theoretical foundations of testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 51-54.

[Miller, 1977e] Miller, Edward F. Notes on tools and techniques of testing. Program Testing Techniques (Miller, ed.). Long Beach, CA: IEEE Computer Society, 1977, p. 107-111.

[Miller, 1977f] Miller, Edward F. Program testing: art meets theory. Computer 10(7): 42-51, 1977.

[Miller, 1977g] Miller, Edward F. Program testing techniques. Long Beach, CA: IEEE Computer Society, 1977.

[Miller, 1977h] Miller, Edward F. Program testing tools -- a survey. MIDCON/77 Proc, Chicago, 1977.

[Miller, 1977i] Miller, Edward F. Toward automated software testing: problems and payoffs. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 342-46.

[Miller, 1978] Miller, Edward F. Program testing: An overview for managers. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 114-19.

[Miller, 1978a] Miller, E. Introduction to software testing. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 3-14.

[Miller, 1978b] Miller, Edward F. Program testing. Computer 11(4): 10-12, 1978.

[Miller, 1978c] Miller, Edward F. Program testing technology in the 1980's. The Oregon Report: Proc of the Conf on Computing in the 1980's. Long Beach, CA: IEEE Computer Society, 1978, p. 72-79.

[Miller, 1978d] Miller, Edward F. et al. Structural techniques of program validation. Tutorial: Software Testing and Validation Techniques, N.Y.: IEEE, 1978, p. 262-65.

[Moranda, 1978] Moranda, Paul B. Limits to program testing with random number inputs. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 521-26.

[Mullin, 1977] Mullin, Frank J. Software test management. Intl Computer Software and Applications Conf, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 321-6.

[Myers, 1976] Myers, Glenford J. Software Reliability: Principles and Practices. N.Y.: Wiley-Interscience, 1976.

[Myers, 1978] Myers, Glenford J. A controlled experiment in program testing and code walkthrough/inspections. Communications of the ACM 21(9): 760-68, 1978.

[Myers, 1979] Myers, Glenford J. The Art of Software Testing, N.Y.: Wiley-Interscience, 1979.

[Osterweil, 1976] Osterweil, L. J.; Fosdick, L. D. Some experience with DAVE -- a Fortran program analyzer. National Computer Conf, AFIPS Conf Proc 45: 909-15, 1976.

[Osterweil, 1976a] Osterweil, Leon J.; Fosdick, Lloyd D. DAVE -- A validation error detection and documentation system for Fortran programs. Software -- Practice and Experience 6(4): 505-25, 1976.

[Osterweil, 1977] Osterweil, L. J. The detection of unexecutable program paths through static data flow analysis. Intl Conf on Computer Software and Applications, 1st (COMPSAC '77), N.Y.: IEEE, 1977, p. 406-13.

[Paige, 1973] Paige, M. R.; Balkovich, E. E. On testing programs. Symp on Computer Software Reliability, N.Y. : IEEE, 1973. p. 23-27.

[Paige, 1974] Paige, Michael; Benson, J. P. The use of software probes in testing Fortran programs. Computer 7(7): 40-47, 1974.

[Paige, 1975] Paige, Michael R. Program graphs, an algebra, and their implication for programming. IEEE Trans on Software Engineering SE-1(3): 286-91, 1975.

[Paige, 1978] Paige, Michael R. An analytical approach to software testing. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 527-32.

[Panzl, 1976] Panzl, David J. Test procedures: a new approach to software verification. Intl Conf on Software Engineering, 2nd, Long Beach, CA: IEEE Computer Society, 1976, p. 477-85.

[Panzl, 1978] Panzl, David J. Automatic software test drivers. Computer 11(4): 44-50, 1978.

[Panzl, 1978a] Panzl, David J. Automatic revision of formal test procedures. Intl Conf on Software Engineering, 3rd, N.Y.: IEEE, 1978, p. 320-26.

[Panzl, 1978b] Panzl, David J. A language for specifying software tests. National Computer Conf, AFIPS Conf Proc 47: 609-19, 1978.

[Persch, 1978] Persch, Guido; Winterstein, Georg. Symbolic interpretation and tracing of PASCAL programs. Intl Conf on Software Engineering, 3rd, N.Y.: IEEE, 1978, p. 312-19.

[Peterson, 1976] Peterson, R. J. TESTER/1: an abstract model for the automatic synthesis of program test case specifications. Symp on Computer Software Engineering, Proc, N.Y.: Polytechnic Press, 1976 p. 465-84.

[Pimont, 1976] Pimont, Simone; Rault, Jean-Claude. A software reliability assessment based on a structural and behavioral analysis of programs. Intl Conf on Software Engineering, 2nd, Long Beach, CA: IEEE Computer Society, 1976, p. 486-91.

[Ramamoorthy, 1973] Ramamoorthy, C. V.; Meeker, R. E.; Turner, J. Design and construction of an automated software evaluation system. Symp on Computer Software Reliability, N.Y.: IEEE, 1973, p. 28-37.

[Ramamoorthy, 1975] Ramamoorthy, C. V.; Kim, K. H.; Chen, W. T. Optimal placement of software monitors aiding systematic testing. IEEE Trans on Software Engineering SE-1(4): 403-11, 1975.

[Ramamoorthy, 1975a] Ramamoorthy, C. V.; Ho, S. F. Testing large software with automated software evaluation systems. IEEE Trans on Software Engineering SE-1(1): 46-58, 1975.

[Ramamoorthy, 1976] Ramamoorthy, C. V.; Ho, S. F.; Chen, W. T. On the automated generation of program test data. IEEE Trans on Software Engineering SE-2(4): 293-300, 1976.

[Reifer, 1977] Reifer, Donald J.; Trattner, Stephen. A glossary of software testing tools and techniques. Computer 10(7): 52-60, 1977.

[Rubey, 1975] Rubey, Raymond J.; Dana, Joseph A.; Biche, Peter W. Quantitative aspects of software validation. IEEE Trans on Software Engineering SE-1(2): 150-55, 1975.

[Sande, 1975] Sande, G. Program execution profiles. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 325-26.

[Schneidewind, 1977] Schneidewind, N. F. The use of simulation in the evaluation of software. Computer 10(4): 47-53, 1977.

[Schneidewind, 1979] Schneidewind, Norman F. Application of program graphs and complexity analysis to software development and testing. IEEE Trans on Reliability R-28(3): 192-98, 1979.

[Stillman, 1975] Stillman, Rona B. Fortran analysis by simple transforms. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 318-24.

[Stucki, 1972] Stucki, Leon G. A prototype automatic testing tool. National Computer Conf, AFIPS Conf Proc 41: 829-36, 1972.

[Stucki, 1973] Stucki, Leon G. Automatic generation of self-metric software. IEEE Symp on Computer Software Reliability, N.Y.: IEEE, 1973, p. 94-100.

[Stucki, 1975] Stucki, Leon G.; Foshee, Gary L. New assertion concepts for self-metric software validation. Intl Conf on Reliable Software, Proc, N.Y.: IEEE, 1975, p. 59-71.

[Stucki, 1975a] Stucki, Leon G. Tools: lessons learned -- new strategies. Computer Science and Statistics: Symp on the Interface, 8th, Los Angeles, CA: UCLA Health Sciences Facility, 1975, p. 313-17.

[Stucki, 1977] Stucki, Leon G. New directions in automated tools for improving software quality. Current Trends in Programming Methodology (R.T. Yeh, ed.). Englewood Cliffs, N.J.: Prentice-Hall, 1977, Vol. II, p. 80-111.

[Tausworthe, 1977] Tausworthe, Robert C. Standardized Development of Computer Software. Englewood Cliffs, N.J.: Prentice-Hall, 1977.

[Taylor, 1978] Taylor, Richard N.; Osterweil, L. J. A facility for verification, testing, and documentation of concurrent process software. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 36-41.

[Van Tassel, 1978] Van Tassel, Dennie. Program Style, Design, Efficiency, Debugging, and Testing. 2d ed. Englewood Cliffs, N.J.: Prentice-Hall, 1978.

[Voges, 1980] Voges, Udo; Gmeiner, Lothar; Amschler von Mayrhausen, Anneliese. SADAT -- an automated test tool. IEEE Trans on Software Engineering SE-6(3): 286-90, 1980.

[Walsh, 1977] Walsh, Dorothy A. Structured testing. Datamation 23(7): 111-18, 1977.

[Workshop, 1979] Workshop report: software testing and test documentation. Computer 12(3): 98-107, 1979.

[Yau, 1978] Yau, S. S.; Chen, F. C.; Yau, K. H. An approach to real-time control flow testing. Intl Computer Software and Applications Conf, 2nd (COMPSAC '78), N.Y.: IEEE, 1978, p. 163-68.

[Yourdon, 1975] Yourdon, Edward. Techniques of Program Structure and Design. Englewood Cliffs, N.J.: Prentice-Hall, 1975.

[Yourdon, 1979] Yourdon, Edward. Structured walkthroughs. 2d ed. Englewood Cliffs, N.J.: Prentice-Hall, 1979.

APPENDIX A

The following pages contain sample documents of a fictitious system design for a dry cleaning establishment inventory-billing system. These documents are not intended to be complete. They are used for illustrative purposes in support of Chapter IV.

CONTENTS:

CUSTOMER REQUEST FOR PROPOSAL
Figure 21

A large dry cleaning establishment has decided to install an on-line system for better control of materials coming in and going out. For each customer, it would like to keep track of materials brought in, day brought, day promised, and any special instructions. In addition, it would like to be able to record a customer's history, since regular customers have charge accounts, and would like to be able to automatically have bills sent to these customers every other month. It would also like to be able to flag delinquent customers so that they will not be allowed to make any charges when bringing in clothing. When a customer pays a bill, it would like to be able to generate an itemized receipt. Some of the reports used by management include:

1. a list of items promised by a certain day

2. a separate report of those items requiring special attention

3. a list of regular customers, with their addresses, for mailing purposes

4. a list of delinquent customers

There are 10 locations scattered throughout a large metropolitan area, each handling an average of 200 customers/day with an average of 3 articles of clothing/customer. The clothing is collected at these locations and shipped to a central site, where it is cleaned, pressed, and repaired. To keep the number of items at each store at a minimum, the clothing is kept at the central site until the evening before promised to the customer, when it is returned to the location.

Currently the clothing is sorted into large bins at the central site by date promised. Those items requiring special attention must be kept separately. Slips are attached to the clothing with the customer's name and the store location, but when a slip is lost, there is no way of identifying the article. Billing is currently done manually, and several patrons run up large accounts. There is no means of knowing who those people are at the time of service.

GENERAL SYSTEM DESIGN
Figure 22

The following functional requirements have been identified as being necessary to the design of an on-line system for the dry cleaning establishment:

1. Data base of current files and customer histories

    1.1. Customer information should include address, account number, billing history, and unpaid items charged to that customer (both past and current)

    1.2. Item information should include type of garment, number of pieces, color, date brought, date promised, store location, price, and special instructions

2. On-line functions

    2.1. The system should be capable of being searched interactively by customer name, account number, or item identifier

    2.2. While searching on-line, the system should notify the user if the customer is delinquent

    2.3. When an item look-up is made, the system should display a report of all the information relevant to that item

    2.4. The user (store clerk) should be able to enter a new customer

    2.5. When an order is placed, the system should either locate the customer or create a new record, and then prompt the user for item information until all items have been entered. A slip with an item identifier should be generated for each item entered. The next item identifier will be automatically calculated by the system.

3. Batch functions

    3.1. Bills should be automatically calculated and generated every month, on a cycling basis (so many bills prepared per day)

3.2. An "Item Promised" Report should be prepared at least one day prior to the date promised.

3.3. A report of items requiring special attention should be prepared at least two days prior to the date promised.

DETAILED SYSTEM DESIGN
Figure 23

1. Data base design

1.1. Records

The data base will consist of four records, including:

a. Store-record, including the store identification number.

b. Item-record, including the item identifier, item type (i.e. suit, sweater, pants, etc.), color, number of pieces, date brought in, date promised, price, and any special instructions.

c. Customer-record, including the customer name, address, phone number, a history of payments and current balance, and the number of bills sent for the current balance.

d. Account-record, including the customer account number.

1.2. Linkages

a. One-to-many relationship between store and item.

b. One-to-many relationship between customer and item.

c. One-to-one relationship between customer and account number.

1.3. Data elements

Record	Element	Length	Type
Store	Store-number *	2	N
Item	Item-ID *	10	N
	Item-type	3	A/N
	Pieces	2	N
	Color	4	A/N
	Date-brought	6	N
	Date-promised	6	N
	Price	4 (2.2)	N
	Special-Instr	50	A/N
Customer	Customer-Name *	30	A/N
	Customer-Address	20	A/N
	Customer-City	15	A/N
	Customer-State	2	A
	Customer-Zip	5	N
	Customer-Phone	12	A/N
	Payments	8 (6.2)	N
	Curr-Bal	8 (6.2)	N
	Bills-sent	2	N
Account	Acct-Num *	8	N

* = search keys

1.4. Required capabilities

a. Searching can be done when only customer name is known.

b. A single item look-up can be made.

c. A listing of all items due on a certain day can be produced, in ascending order.

d. A listing of all items due on a certain day for an individual store can be produced, in ascending order.

e. A listing of all credit customers can be produced.

2.0. On-line functions

:

:

:

2.5. Item entry

2.5.1. Customer look-up

a. The system will prompt for the customer's name. The user should enter the name of the customer, last name first.

b. If the name is found:

(1) the system will display that customer's address, phone number, and (when applicable) account number and balance due.

(2) if the balance due is over $50 and a second bill has been sent, the system will display a delinquent customer message.

(3) the system will prompt the user with "Same Customer ?". If the user enters "Y", the system enters the item entry mode (2.5.2). Otherwise, it will prompt the user for more name information and return to 2.5.1.

c. If the name is not found:

(1) The system enters the "New Customer" mode, and will prompt the user for address and phone number.

(2) the system will ask if the new customer would like to establish a charge account. If the user enters "Y", the next available account number will be automatically generated and linked to the customer's record.

(3) the system then enters the item entry mode (2.5.2)

2.5.2. Item entry

a. The system will prompt the user for item type. The user will enter the type of garment (i.e. suit, sweater, pants, etc.).

b. The system will then prompt for number of pieces, color, date promised by, and special instructions.

c. The system will obtain the present date from the computer.

d. Price will be calculated based on the item type and special instructions. First, a table look-up will be made to obtain the set price of an item. If there are any special instructions, a table look-up will be made (by code) for the price, and that will be added to the set price of the item.

e. After all information has been stored within the item record, the system will prompt the user for the next item, returning to 2.5.2.a.

f. To exit, the user will type "end" when the system prompts for the next item, and will be returned to the general on-line mode (2.0).

UPDATE PROGRAM LISTING
Figure 24

```
00100   IDENTIFICATION DIVISION.
00200   PROGRAM-ID. PROGRAM-1.
00250
00300   DATA DIVISION.
00400
00500   SCHEMA SECTION.
00600       INVOKE SUB-SCHEMA CLEAN OF SCHEMA LAUND.
00700
00800   WORKING-STORAGE SECTION.
00900
01000   77  WISH                        PIC X.
01100   77  CUST-NAME                   PIC X(30).
01200   77  CHARGE                      PIC X.
01300   77  ACCOUNT-NUMBER              PIC 9(08).
01400   77  SAME-CUSTOMER               PIC X.
01500   77  ITEM-NO                     PIC 9(10).
01600
01700   01  ERROR-FLAG                  PIC 9 VALUE 0.
01800       88  END-OF-CUSTOMER-SET           VALUE 1.
01900       88  FOUND                         VALUE 2.
02000       88  DONE                          VALUE 3.
02100
02200   01  TODAYS-DATE.
02300       03  THE-DATE                PIC 9(06).
02400       03  FILLER                  PIC 9(06).
02500
02600   PROCEDURE DIVISION.
02700   A-MAIN.
02800       OPEN ALL USAGE-MODE IS UPDATE.
02900       DISPLAY SPACE, DISPLAY SPACE, DISPLAY SPACE,
03000       DISPLAY SPACE, DISPLAY SPACE, DISPLAY SPACE.
03100       DISPLAY "STORE => " WITH NO ADVANCING.
03200       ACCEPT STORE-NUMBER.
03300       FIND STORE-RECORD.
03400       IF ERROR-COUNT > 0,
03500               STORE STORE-RECORD.
03600   A-1-WISH.
03700
03800       DISPLAY SPACE, DISPLAY SPACE.
03900       DISPLAY"         1  ORDER ENTRY".
04000       DISPLAY "        2  PAYMENT".
04100       DISPLAY "        3  ITEM INQUIRY".
04200       DISPLAY SPACE, DISPLAY SPACE.
04300       DISPLAY "        => " WITH NO ADVANCING.
04400       ACCEPT WISH.
```

```
04500       IF WISH = "E", CLOSE ALL, STOP RUN.
04600       IF WISH < "1" OR WISH > "3"
04700               DISPLAY "ERROR. TRY AGAIN.",
04800               GO TO A-1-WISH.
04900       IF WISH = "1" PERFORM B-ORDER.
05000       IF WISH = "2" PERFORM C-PAYMENT.
05100       IF WISH = "3" PERFORM D-ITEM.
05200       GO TO A-1-WISH.
05300
05400   B-ORDER SECTION.
05500
05600   B-ENTER.
05700
05800       DISPLAY "CUSTOMER NAME  => " WITH NO ADVANCING.
05900       ACCEPT CUST-NAME.
06000       PERFORM B-1-FIND UNTIL FOUND
06050               OR END-OF-CUSTOMER-SET.
06100       IF FOUND, PERFORM B-4-OLD-CUSTOMER.
06200       IF END-OF-CUSTOMER-SET,
06300               PERFORM B-2-NEW.
06400       PERFORM B-3-ITEM-ENTRY UNTIL DONE.
06500
06600   B-EXIT.
06700       EXIT.
06800
06900   B-1-FIND SECTION.
07000
07100   B-1-ENTER.
07200       MOVE CUST-NAME TO CUSTOMER-NAME.
07300       FIND CUSTOMER-RECORD.
07400       IF ERROR-COUNT > 0,
07500               MOVE 1 TO ERROR-FLAG,
07600               GO TO B-1-EXIT.
07700       GET.
07800               MOVE 2 TO ERROR-FLAG.
07900
08000   B-1-EXIT.
08100
08200       EXIT.
08300
08400   B-2-NEW SECTION.
08500
08600   B-2-ENTER.
08700
08800       DISPLAY "ADDRESS => " WITH NO ADVANCING.
08900       ACCEPT CUSTOMER-ADDRESS.
09000       DISPLAY "CITY => " WITH NO ADVANCING.
09100       ACCEPT CUSTOMER-CITY.
09200       DISPLAY "STATE => " WITH NO ADVANCING.
```

```
09300         ACCEPT CUSTOMER-STATE.
09400         DISPLAY "ZIP => " WITH NO ADVANCING.
09500         ACCEPT CUSTOMER-ZIP.
09600         DISPLAY "PHONE (XXX-XXX-XXXX) => ".
09700         ACCEPT CUSTOMER-PHONE.
09800     B-2-CHARGE.
09900
10000         DISPLAY "CHARGE ? => " WITH NO ADVANCING.
10100         ACCEPT CHARGE.
10200         IF CHARGE NOT EQUAL "Y",
10300                 GO TO B-2-EXIT.
10400         FIND LAST ACCOUNT-RECORD RECORD OF ACCT-SET SET.
10500         GET.
10700         ADD 1 TO ACCOUNT-NUMBER.
10800         DISPLAY SPACE.
10900         DISPLAY "ACCOUNT NUMBER FOR ",CUSTOMER-NAME,
11000                 " IS ", ACCOUNT-NUMBER.
11100         MOVE ACCOUNT-NUMBER TO ACCT-NUM.
11200         STORE ACCOUNT-RECORD.
11300
11400     B-2-STORE.
11500         STORE CUSTOMER-RECORD.
11600         IF ERROR-COUNT > 0,
11700                 ACCEPT CUSTOMER-NAME,
11800                 GO TO B-2-STORE.
11900
12100     B-2-EXIT.
12200         EXIT.
12300
12400     B-3-ITEM-ENTRY SECTION.
12500
12600     B-3-ENTER.
12700
12800         FIND LAST ITEM-RECORD RECORD OF ITEM-SET SET.
12900         GET.
13000         MOVE ITEM-ID TO ITEM-NO.
13100         ADD 1 TO ITEM-NO.
13200         MOVE ITEM-NO TO ITEM-ID.
13300         DISPLAY "TYPE => " WITH NO ADVANCING.
13400         ACCEPT ITEM-TYPE.
13500         IF ITEM-TYPE = "END",
13600                 MOVE 3 TO ERROR-FLAG,
13700                 GO TO B-3-EXIT.
13800         DISPLAY "PIECES => " WITH NO ADVANCING.
13900         ACCEPT PIECES.
14000         DISPLAY "COLOR => " WITH NO ADVANCING.
14100         ACCEPT COLOR.
14300         MOVE THE-DATE TO DATE-BROUGHT.
14400         DISPLAY "PROMISED BY (MMDDYY) => ".
```

```
14500       ACCEPT DATE-PROMISED.
14600       DISPLAY "SPECIAL INSTRUCTIONS => ".
14700       ACCEPT SPECIAL-INSTR.
14800       PERFORM PRICE-FIND.
14900       STORE ITEM-RECORD.
15000
15100   B-3-EXIT.
15200       EXIT.
15300
15400   B-4-OLD-CUSTOMER SECTION.
15500
15600   B-4-ENTER.
15700
15800       DISPLAY "ADDRESS       : ", CUSTOMER-ADDRESS.
15900       DISPLAY "                ", CUSTOMER-CITY,", ",
16000               CUSTOMER-STATE, " ", CUSTOMER-ZIP.
16100       DISPLAY "PHONE         : ", CUSTOMER-PHONE.
16200       DISPLAY SPACE, DISPLAY SPACE,
16300       DISPLAY "SAME CUSTOMER ? => " WITH NO ADVANCING.
16400       ACCEPT SAME-CUSTOMER.
16500       IF SAME-CUSTOMER = "N",
16600               DISPLAY "NAME MATCHES ANOTHER CUSTOMER."
16700               DISPLAY "ENTER FULLER NAME => ",
16800               ACCEPT CUSTOMER-NAME,
16900               MOVE 1 TO ERROR-FLAG.
17000
17100   B-4-EXIT.
17200       EXIT.
17300
17400   C-PAYMENT SECTION.
17500
17600   C-ENTER.
17700
17800   C-EXIT.
17900       EXIT.
18000
18100   D-ITEM SECTION.
18200
18300   D-ENTER.
18500   D-EXIT.
18600       EXIT.
18700
18800   PRICE-FIND SECTION.
18900   PRICE-ENTER.
19000       IF ITEM-TYPE = "SUI" MOVE 5 TO PRICE.
19100       IF ITEM-TYPE = "SKI" MOVE 2.5 TO PRICE.
19300   PRICE-EXIT.
19400       EXIT.
```

APPENDIX B

The following chart summarizes information on several automated software testing tools, as mentioned in Chapter IV. This is not a comprehensive list, but is meant more to demonstrate the types of tools which can be built or purchased. (Not all tools listed here are available for purchase.)

The National Bureau of Standards is in the process of compiling a software tools data base. Interested persons should contact Raymond C. Houghton of the NBS, Washington, D.C.

TOOL	DEVELOPER/CONTACT	LANGUAGES ACCEPTED	FUNCTIONS
AMPIC	Marilyn S. Fujii Logicon, Inc. P.O. Box 471 San Pedro, CA 90733	Fortran Jovial	Symbolic execution Directed Graph production Path analysis
ANSI FORTRAN CHECKER	Softool Corp. 340 S. Kellogg Ave. Goleta, CA 93017	Fortran	Checks for compliance with ANSI Fortran
ATTEST	Lori A. Clarke Dept. of Computer and Info. Sci. Univ. of Mass. Amherst, Mass. 01003	Fortran	Data flow analysis Symbolic execution Test data generation Path selection
DATAMACS	Management and Computer Services, Inc. Valley Forge, PA 19482	COBOL	Test data generation
DAVE	L.J. Osterweil Dept. of Comp. Sci. Univ. of Colo. Boulder, Colo. 80309	Fortran	Data flow analysis
DISSECT	W. E. Howden Dept. of Math. Univ. of Victoria Victoria, B.C. CANADA	Fortran	Symbolic evaluation

SOFTWARE TESTING TOOLS
Figure 25

FACES	COSMIC Suite 112 Barrow Hall Univ. of Georgia Athens, GA 30602	Fortran	Detects coding errors and unsound coding practices in ANSI Fortran
INSTRUMENTER	Softool Corp. 340 South Kellogg Ave. Goleta, CA 93017	Fortran	Test coverage monitor
JAVS	Carolyn Gannon General Research Corp. Santa Barbara, CA 93111 (for:RADC/ISIE, Griffiss AFB, Rome, N.Y. 13441)	Jovial	Directed graph production Probe insertion Test coverage monitor
NBS ANALYZER	Gordon Lyon NBS Inst. for Comp. Sci. and Tech. Tech Bldg. A 265 Wash, DC 20234	Fortran	Frequency statistics on constructs and on code usage
NODAL	TRW, Inc. One Space Park Redondo Beach, CA 90278	Fortran	Execution frequency statistics Probe insertion
PACE	TRW Software Quality Assurance One Space Park Redondo Beach, CA 90278	Fortran	Branch testing Test effectiveness measurement

SOFTWARE TESTING TOOLS
Figure 25

PET	Z. Jelinski McDonnell Douglas 5301 Bolsa Ave. Huntington Beach, CA 92647	Fortran	Lexical analysis Statement testing Branch testing Subroutine timing
RXVP	General Research Corp. Santa Barbara, CA 93111	Fortran	Branch testing Statement testing Test coverage monitor Probe insertion
SADAT	M. Seifert Kernforschungszentrum Karlsruhe GmbH Inst. fur Daten verarbeitung in der Technik Postfach 3640, D-7500 Karlsruhe Federal Rep. of Germany	Fortran	Lexical analysis Directed graph production Data flow analysis Test case generation Symbolic execution Path selection
SAP	NASA/Goddard Space Flight Center Greenbelt, MD 20771 (Contact: COSMIC)	Fortran	Lexical analysis to produce statement occurence frequencies
XPEDITER	Edmond F. Harris Application Devt. Syst, Inc. 1530 Meridian Ave. San Jose, CA 95125	Fortran COBOL Assembler	Test execution monitor Tracing Module driver Module simulator

SOFTWARE TESTING TOOLS
Figure 25

VITA

The author was born on March 27, 1950, in Philadelphia, Pa., to John J. and Helen Aqua. She completed high school in 1968, and attended Kirkland College, Clinton, N.Y. from 1968 to 1972. She received her Bachelor of Arts degree in 1972, with a major in mathematics.

From 1972 to 1975, the author served as a Peace Corps volunteer in Nepal, where she taught math and science in a remote Nepalese hill village.

Upon returning to the United States (1976), she attended the Graduate School of Library and Information Science, University of Pittsburgh, and received her Master of Library Science degree in January of 1977.

In February of that year, she began full-time employment as a reference librarian in Mart Science and Engineering Library, Lehigh University, where she remained until June of 1979. During that period, she co-authored a book with Christine Roysdon entitled "American Engineers of the 19th Century: a biographical index" (N.Y.: Garland Pub Co, 1978). She also served as an active member of the Lehigh University Forum.

In June of 1978, the author began part-time studies as a graduate student in the Industrial Engineering Department of Lehigh University. One year later (June, 1979), she became a full-time student, simultaneously assuming the responsibilities of a research assistant for the Department of Energy data base project.

Upon completion of this degree, she will be pursuing a career with Bell Telephone Laboratories.